USENIX Proceedings of the 8th USENIX Security Symposium (Security '99) *Washington, D.C., USA, August 1999*

conference

proceedings

Eighth USENIX Security Symposium (Security '99)

Washington, D.C., USA
August 23–26, 1999

Sponsored by

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In Cooperation with the
CERT Coordination Center

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $27 for members and $35 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past USENIX Security Proceedings

Security VII	January 1998	San Antonio, Texas, USA	$27/35
Security VI	July 1996	San Jose, California, USA	$27/35
Security V	June 1995	Salt Lake City, Utah, USA	$27/35
Security IV	October 1993	Santa Clara, California, USA	$15/20
Security III	September 1992	Baltimore, Maryland, USA	$30/39
Security II	August 1990	Portland, Oregon, USA	$13/16
Security	August 1988	Portland, Oregon, USA	$7/7

ISBN 1-880446-28-6

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Eighth USENIX Security Symposium (Security '99)

August 23–26, 1999
Washington, D.C., USA

Conference Organizers

Program Chair

Win Treese, *Open Market, Inc.*

Program Committee

Fred Avolio, *Avolio Consulting*
Crispin Cowan, *Oregon Graduate Institute*
Jim Duncan, *Cisco Systems, Inc.*
Carl Ellison, *Intel Corporation*
Daniel Geer, *CertCo, Inc.*
Peter Gutmann, *University of Auckland*
Trent Jaeger, *IBM*
Wolfgang Ley, *Sun Microsystems*
Alain Mayer, *Lucent Technologies, Bell Laboratories*
Christoph Schuba, *Sun Microsystems Laboratories*
Peter Trei, *Security Dynamics, Inc.*
Dan Wallach, *Rice University*

Invited Talks Coordinator

Aviel Rubin, *AT&T Labs—Research*

The USENIX Association Staff

External Reviewers

Rich Ankney	Sandeep Kumar
Steve Beattie	Thomas Markson
Germano Caronni	Rich Salz
Don Davis	Perry Wagle
Amit Gupta	Marcel Waldvogel
Heather Hinton	Nathalie Weiler
Sitaram Iyer	

Contents

The 8th USENIX Security Symposium

August 23–26, 1999
Washington, D.C., USA

Index of Authors

Message from the Program Chair

I am pleased to welcome you to the 8th USENIX Security Symposium. This conference is a unique gathering for those interested in computer and network security, spanning a broad range of both theory and practice. The papers here represent that full range.

It is almost trite today to say that computer security is a popular topic, with stories appearing almost daily in newspapers, magazines, and television programs. Yet behind the faddish talk is a serious reality: as we enter the 21st century, security is critically important for computers and networks. The reality is both better and worse than is reported in the press—better because many organizations do pay attention to (and money for!) security, and worse because we know there are still so many attacks and problems to be solved.

The papers presented here all contribute to solving those problems. Each advances the state of the art in some way, whether describing a security architecture, implementation experience, or ways to make security easier for the end user to use and understand. The session titles alone—PDAs, Cages, Keys, Security Practicum, Access Control, and Potpourri—tell the story of the broad group of security problems and solutions addressed here.

The program committee considered 67 papers, a new record for this symposium. From those, the committee selected 18 for the refereed papers track. As a whole, the submitted papers were of very high quality, and the committee wishes we had time for more papers to be presented. We had an especially large number of submissions from outside the U.S.A., a trend we hope will continue in future symposia.

I would like to thank those who have contributed to organizing this symposium. The program committee and the external readers dedicated their time to reading and evaluating the submitted papers. The USENIX staff, as always, has been unfailingly efficient and helpful at every turn. Most especially, I would like to thank the authors of the submitted papers, without whom we would have no symposium.

I hope you find this conference as interesting and exciting as I expect it to be.

Win Treese
Program Chair

The Design and Analysis of Graphical Passwords

Ian Jermyn
New York University
jermyn@cs.nyu.edu

Alain Mayer, Fabian Monrose, Michael K. Reiter
Bell Labs, Lucent Technologies
{alain,fabian,reiter}@research.bell-labs.com

Aviel D. Rubin
AT&T Labs—Research
rubin@research.att.com

Abstract

In this paper we propose and evaluate new *graphical* password schemes that exploit features of graphical input displays to achieve better security than text-based passwords. Graphical input devices enable the user to decouple the *position* of inputs from the *temporal order* in which those inputs occur, and we show that this decoupling can be used to generate password schemes with substantially larger (memorable) password spaces. In order to evaluate the security of one of our schemes, we devise a novel way to capture a subset of the "memorable" passwords that, we believe, is itself a contribution. In this work we are primarily motivated by devices such as personal digital assistants (PDAs) that offer graphical input capabilities via a stylus, and we describe our prototype implementation of one of our password schemes on such a PDA, namely the Palm Pilot$^{\text{TM}}$.

1 Introduction

For the vast majority of computer systems, passwords are the method of choice for authenticating users. It is well-known, however, that passwords are susceptible to attack: users tend to choose passwords that are easy to remember, and often this means that they are also easy for an attacker to obtain by searching for candidate passwords. In one case study of 14,000 Unix passwords, almost 25% of the passwords were found by searching for words from a carefully formed "dictionary" of only 3×10^6 words [12]. This relatively high success rate is not unusual despite the fact that there are roughly 2×10^{14} 8-character passwords consisting of digits and upper and lower case letters alone.

In this paper we explore an approach to user authentication that generalizes the notion of a textual password and that, in many cases, improves the security of user authentication over that provided by textual passwords. We design and analyze *graphical passwords*, which can be input by the user to any device with a graphical input interface. A graphical password serves the same purpose as a textual password, but can consist, for example, of handwritten designs (drawings), possibly in addition to text. The devices by which we are primarily motivated are "personal digital assistants" (PDAs) such as the Palm Pilot$^{\text{TM}}$, Apple Newton$^{\text{TM}}$, Casio Cassiopeia E-10$^{\text{TM}}$, and others, which allow users to provide graphics input to the device via a stylus. More generally, graphical passwords can be used whenever a graphical input device, such as a mouse, is available.

To the best of our knowledge, the notion of a "graphical password" is due to Blonder [4]. That work proposed a password scheme in which the user is presented with a predetermined image on a visual display and required to select one or more predetermined positions ("tap regions") on the displayed image in a particular order to indicate his or her authorization to access the resource. Beyond this proposal, however, [4] did not further explore the power of graphical passwords or argue security for its particular proposal.

In this paper we considerably advance the theory and practice of graphical passwords. We take as a main criterion the need to evaluate graphical passwords' security relative to that of textual passwords. We design two graphical password schemes that we believe to be more secure than textual passwords (and more secure than the scheme of [4]), and we employ novel analysis techniques to make this argument. Moreover, we describe our implementation of one of our graphical password schemes on the Palm Pilot.

The graphical password schemes that we propose derive their strength from the following observation: a graphical interface for providing input enables the user to decouple the *positions* of the inputs from their *temporal order*. This is in contrast to textual passwords input via a keyboard: here, the temporal order in which the user types characters uniquely determines their position in the password. However, in a graphical password, e.g., consisting of several drawn lines, the final position of each line can be determined independently of the temporal order in which the lines are drawn. We show that this independence between input position and order can be used to build interesting new password schemes, and in some cases obtain authentication that is convincingly stronger than textual passwords but not significantly harder to remember.

The first graphical password scheme builds directly on textual password schemes, by enhancing the input of textual passwords using graphical techniques. In this case, if we assume the same underlying distribution on the choice of the password, the graphical password is at least as strong as the textual password that underlies it, and even a conservative estimate of the variations introduced by the graphical input yields a substantial improvement in strength over the purely textual version. We propose and implement a second scheme, called "draw a secret" (DAS), which is purely graphical; the user draws a secret design (the password) on a grid. Here, to argue an improvement over textual passwords, we define a class of DAS passwords that, we believe, captures a small subset of the memorable ones. This class consists of those passwords that can be generated by a short program in a simple grid-based language. We do not argue that every memorable password has a short program to describe it, but that passwords describable by short programs are memorable. We show that even this subset of memorable DAS passwords is larger than the dictionaries of textual passwords to which a high percentage of passwords typically belong.

Throughout this paper we focus on graphical passwords that are exactly repeatable by the user. This distinguishes our work from all works on graphical pattern recognition of which we are aware (see Section 4), where it suffices for the device to recognize an input as being "sufficiently similar" to—but not necessarily the same as—a previously stored input. Because pattern recognition schemes require the storage of (some representation of) the plaintext password on the device, the password is vulnerable to an attacker who captures and probes the device. In contrast, because graphical passwords are repeatable, our schemes can derive a secret key, e.g., to encrypt and decrypt files, without need to store the password on the device. This protects both the password and the encrypted content from the attacker if the device falls into the attacker's hands.

The rest of this paper is outlined as follows: In Section 2, we present textual passwords with graphical assistance. In Section 3, we proceed to purely graphical passwords with a scheme called "draw-a-secret" (DAS). Section 3.2 shows our design and implementation of a memo pad encryption scheme based on DAS. Section 3.3 proposes novel ways to analyze and estimate the security of DAS and graphical passwords in general. In Section 4 we overview other password schemes, unrelated to graphical passwords, but putting our work in a larger context. Finally, Section 5 concludes.

2 Textual Passwords with Graphical Assistance

In this section we present a password selection and input scheme which uses textual passwords augmented by some minimal graphical capabilities that enable the decoupling of temporal order of input and the position in which characters are input. This scheme is interesting because it simply demonstrates the power of graphical input abilities while yielding a scheme that is convincingly stronger than textual passwords are today.

We start by defining a normal, k-character textual password as a total function $\pi : \{1, \ldots, k\} \to \mathcal{A}$, where $\mathcal{A}$ is the set of allowed characters for the textual password. Intuitively, the domain of π denotes the temporal order of inputs, so that the user first enters $\pi(1)$, then $\pi(2)$, and so on. That is, for a password "`tomato`", we have $\pi(1) = \texttt{t}$, $\pi(2) = \texttt{o}$, $\pi(3) = \texttt{m}$, $\pi(4) = \texttt{a}$, $\pi(5) = \texttt{t}$, and $\pi(6) = \texttt{o}$.

Now suppose that the user is presented with a simple graphical input display consisting of, say, eight positions into which to enter a textual password, as illustrated in Figure 1. In this figure, step 0 is the initial row of blanks, and steps 1–6 indicate the temporal order in which the user fills in the blanks; i.e., $\pi(i)$ is entered in row i. The password can be placed in the "normal", left-to-right positions as shown in Figure 1a. Due to the graphical nature of the input interface, however, the user could enter the password in other positions, as well. For example, Figure 1b shows a modification in which the user enters the password in a left-to-right manner, but starting from a different initial position than the leftmost. Figure 1c shows entering the password in an "outside-in" strategy. And, of course, these variations can be combined in the obvious way, as shown in Figure 1d.

Formally, a k-character graphical password in this scheme can be defined by a total function $\pi' : \{1, \ldots, k\} \to \mathcal{A} \times \{1, \ldots, m\}$, where $m \geq k$ is the number of positions into which characters can be entered ($m = 8$ in Figure 1). If $\pi'(i) = (c, j)$, then this means that the i-th entry (temporally) is the character c in position j. A conventional textual password π, entered in the standard left-to-right way, can be expressed in this scheme as a graphical password π' where $\pi'(i) = (\pi(i), i)$. But as shown in Figure 1, more generally we can have variations π' in which $\pi'(i) = (\pi(i), j)$ and $i \neq j$. In fact, it is easy to see that each k-character conventional password π yields $m!/(m-k)!$ graphical passwords π', and indeed this is the factor by which the size of the graphical password space exceeds the k-character conventional password space. This can be a relatively large number: e.g., for $k = 8$ and $m = 10$, this factor is approximately 2×10^6.

Of course, there are far fewer than 2×10^6 variations of each 8-character password that are memorable for human users. However, it is easy to derive a convincing lower bound on the improvement this achieves over a conventional password scheme. It is conservative to assume that the m positional rotations of a password, plus perhaps a handful of others (e.g., reversal, outside-in, inside-out, evens-then-odds, odds-then-evens), and combinations thereof, are memorable, because the choices of position involved in these cases can be derived from simple algorithms that are more memorable than the positions themselves. (We will return to this characteristic of memorability in the next section.) The attacker's work load will thus be increased by a factor of at least m. An important feature of this scheme is that it is at least as strong as the initial textual password that was chosen by the user, assuming that users do not reduce the size of the space of character sequences that they choose in response to the need to remember a positional order.

There are a number of steps that we can take to make this scheme more usable. First, to maximize the ease of inputting passwords with varied position, each character should be echoed once the user places it in a position, at least with a nondescript character (e.g., "$*$") but preferably with the letter itself. This is a departure from most password-input interfaces, which echo at most a nondescript character in order to protect the password from onlooking persons. However, for the platforms by which we are primarily motivated, i.e., hand-held PDAs such as the Palm Pilot, it is much easier to shield the screen from onlookers entirely. Going further, the interface might allow the user to first enter the password "normally" (left-to-right), and then drag each character to its final position.

Inevitably, there are numerous variations on the scheme presented here. One direction includes arranging the k input positions in some other way than a straight line (e.g., a grid), to promote other variations in position. Rather than pursuing these options here, we instead explore a purely graphical approach.

3 The Draw-a-Secret (DAS) Scheme

In this section we present a purely graphical password selection and input scheme, which we call "draw a secret" (DAS). In this scheme, the password is a simple picture drawn on a grid. This approach is alphabet independent, thus making it equally accessible for speakers of any language. Users are freed from having to remember any kind of alphanumeric string.

```
0. _ _ _ _ _ _ _ _
1. t _ _ _ _ _ _ _
2. t o _ _ _ _ _ _
3. t o m _ _ _ _ _
4. t o m a _ _ _ _
5. t o m a t _ _ _
6. t o m a t o _ _
```

(a) Left-to-right

```
0. _ _ _ _ _ _ _ _
1. _ _ _ _ _ _ _ t
2. o _ _ _ _ _ _ t
3. o m _ _ _ _ _ t
4. o m a _ _ _ _ t
5. o m a t _ _ _ t
6. o m a t o _ _ t
```

(b) Rotated left

```
0. _ _ _ _ _ _ _ _
1. t _ _ _ _ _ _ _
2. t _ _ _ _ o _ _
3. t m _ _ _ o _ _
4. t m _ _ a o _ _
5. t m t _ a o _ _
6. t m t o a o _ _
```

(c) Outside-in

```
0. _ _ _ _ _ _ _ _
1. _ _ _ _ _ _ _ t
2. _ _ _ _ o _ _ t
3. m _ _ _ o _ _ t
4. m _ _ a o _ _ t
5. m t _ a o _ _ t
6. m t o a o _ _ t
```

(d) A more complex example

Figure 1: Variations on inputting `tomato`. The word `tomato` can be input in the "normal" left to right manner as shown in (a). Step 0 is the initial row of blanks, and steps 1–6 indicate the temporal order in which the user fills in the blanks. In addition, however, the user can vary the position of the letters in `tomato`. Figure (b) demonstrates shifting the input left by one, (c) represents an outside-in input strategy, and (d) is the combination of these.

The most compelling reason for exploring the use of a picture-based password scheme is that humans seem to possess a remarkable ability for recalling pictures (i.e., line drawings and real objects). The "picture effect", that is, the effect of pictorial and object representations on a variety of measures of learning and memory has been studied for decades [7, 27, 25, 30, 5]. Cognitive scientists have shown that there is a substantial improvement of performance in recall and recognition with pictorial representations of to-be-remembered material than for verbal representations.

Superiority in recall of objects over words in immediate recall and over short retention intervals has been demonstrated through a number of experiments. Empirical evidence of the power of pictures over words dates back to the 1800s; experiments performed by Calkins [7] showed the recall of words declining by 50% or more over a 72 hour retention interval, and recall of objects dropping by less than 20% over the same period. Studies exhibiting strikingly high differences in memory recall of pictures over words have since been replicated on numerous occasions [27, 30, 22, 6]. Some theories that have been proposed to explain these experimental results are outlined in Appendix A.

3.1 Password Selection and Input

Consider an interface consisting of a rectangular grid of size $G \times G$. Each cell in this grid is denoted by discrete rectangular coordinates $(x, y) \in [1..G] \times [1..G]$. Suppose that the the user is given a stylus with which she can draw a design on this grid. The drawing is then mapped to a sequence of coordinate pairs by listing the cells through which the drawing passes in the order in which it passes through them, with a distinguished coordinate pair inserted in the sequence for each "pen up" event, i.e., whenever the user lifts the stylus from the drawing surface. For example, consider the drawing in Figure 2. Here, the coordinate sequence generated by this drawing is

$$(2,2), (3,2), (3,3), (2,3), (2,2), (2,1), (5,5)$$

where $(5,5)$ is the distinguished "pen up" indicator. If there were a second stroke in this example, then its sequence would be appended to the end of the sequence above, and similarly for subsequent strokes. In this way, we divide the space of possible drawings into equivalence classes, two drawings being equivalent if they have the same encoding, or in other words if they cross the same sequence of grid cells, with the breaks between strokes occurring in the same places.

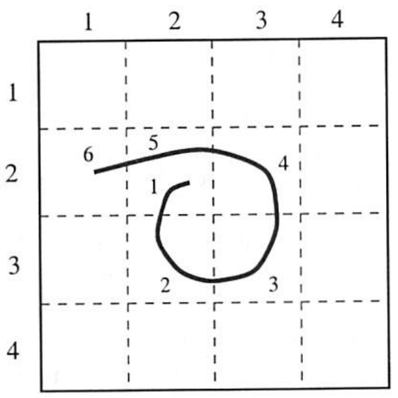


Figure 2: Input of a graphical password on a 4 × 4 grid. The drawing is mapped to a sequence of coordinate pairs by listing the cells in the order which the stylus passes through them, with a distinguished coordinate pair inserted in the sequence whenever the stylus is lifted from the drawing surface.

First we give some terminology. We define the neighbors, $\mathcal{N}_{(x,y)}$, of a cell (x, y) to be the subset of the set of cells $\{(x-1, y), (x+1, y), (x, y-1), (x, y+1)\}$ whose elements exist in the grid. We then define a stroke to be a sequence of cells $\{c_i\}$, in which $c_i \in \mathcal{N}_{c_{i-1}}$, and which does not contain a "pen up" event. A password is then defined to be a sequence of strokes separated by "pen up" events. The length of a stroke is the number of coordinate pairs it contains, while the total length of a password is the sum of the lengths of its component strokes (excluding the "pen up" characters).

As with the scheme of Section 2, this scheme is most viable if the user's strokes are echoed as curves while they are drawn. Again we appeal to the maneuverability of the devices we are targeting (i.e., PDAs) to support the restriction that the user must shield the input display from onlookers.

Our requirement of repeatability constrains the parameters of this scheme. As long as the user's current drawing lies in the same equivalence class as the original drawing, she has successfully repeated a chosen password. In general, this gives the user sufficient tolerance when (involuntarily) varying the drawing, provided that the cells of the grid are not too small. Indeed, this was the purpose of separating the drawings into equivalence classes to begin with. Difficulties might arise however, when the user chooses a drawing that contains strokes that pass too close to a grid-line. In those cases, the user might vary the drawing in such a way as to change the resulting sequence of coordinates. There are at least two solutions to this problem: (1) The user is offered to view the internal representation, depicting the path of cells, when she chooses a password so that she can confirm which cells were actually touched by the drawing. (2) The system does not accept a drawing which contains strokes that are located "too close" to a grid line. In the implementation, described in Section 3.2, we offer both alternatives.

3.2 Application of DAS: An Encryption Tool for a PDA

Our graphical password schemes are motivated primarily by PDAs that offer graphical input capabilities. We now describe our implementation of a memo pad encryption tool for the Palm Pilot that uses a user-input graphical password to derive the encryption key. The Pilot supports a very natural form of graphical input, and as such, provides an ideal platform for implementing the DAS scheme. Either of the schemes of Sections 2 and 3 could be used to enter the password. Here we illustrate our tool using the DAS scheme.

In our tool, an encryption/decryption key is derived from a DAS password (i.e., its sequence of coordinates) as follows: Let $\mathcal{B}$ be a bit string that represents the sequence of coordinates (including the unique "pen up" indicator). The key k is defined as $k = h(\mathcal{B})$ where h is the cryptographic hash function SHA-1. This key derivation assures that two distinct coordinate sequences are transformed (with high probability) into two distinct, fixed-length keys. Triple-DES[1] is then used to encrypt and decrypt data records stored on the PDA, using keys derived from k.

Key selection is as follows: the user is prompted with an empty grid to input the password design. Once the password is entered, k is derived and a pre-defined phrase p is encrypted (as $E_k(p)$) and stored on the PDA. On repeat access, the user is prompted again with the empty grid, upon which she draws the same design. A symmetric key k' is derived and an attempt is made to decrypt $E_k(p)$. If it results in p, then $k' = k$ and the password

[1]Based on Ian Goldberg's port of SSLeay for the Pilot (see `http://www.isaac.cs.berkeley.edu/pilot`).

(and key) is accepted. The user then can proceed to encrypt/decrypt data records. k is deleted from the PDA at the latest when the PDA is powered off.

An adversary who captures the PDA can presumably obtain all of the ciphertext encrypted under k, and since p is either public or stored in plaintext on the device, the adversary has at least one known plaintext/ciphertext pair with which to attack E. For a strong encryption scheme E, however, the best bet for the attacker remains to guess the original password, which, as we will show in Section 3.3, on average is likely to be much harder than if the attacker were faced with attacking a textual password.

The interface for our DAS implementation is shown in Figure 3. Our application shares the database of the `memopad` application, and allows a user to encrypt/decrypt records in the database based on a user specified drawing. The encryption tool for the Palm Pilot is available from `http://cs.nyu.edu/fabian/pilot/gpw.html`.

3.3 Security of the DAS Scheme

We define the information content of a password space as the entropy of the probability distribution over that space given by the relative frequencies of the passwords that users actually choose. Information content is the correct measure for describing difficulty of attack, since it determines the optimal choices to be made when trying different possibilities for a password.

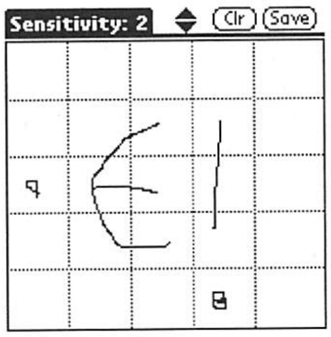
High information content renders a password scheme more or less invulnerable. For example, if users did in fact choose passwords uniformly from the space of all textual passwords, successful attacks would be extremely unlikely. What is it that renders such attacks successful in practice? There are two factors. The first is that in reality users do not choose their passwords uniformly. If we assume that the data collected in Klein's study [12] is representative of the general population, then users in fact use only 10^{-8} of the possible passwords 25% of the time. Such a distribution is highly peaked, and the information content of the textual password space is correspondingly reduced.

However, the fact that users do not pick passwords uniformly is in itself not sufficient to make password guessing attacks successful. The second factor that renders textual passwords vulnerable is that the attacker has significant knowledge of the distribution of user passwords, and can use that knowledge to her advantage. In the case of textual passwords, this knowledge includes information about specific peaks in the distribution (users often choose passwords based on their own name), and information about gross properties (words in the English dictionary are likely to be chosen). Without information about the distribution, an attacker would be no better off than if users were in fact choosing uniformly.

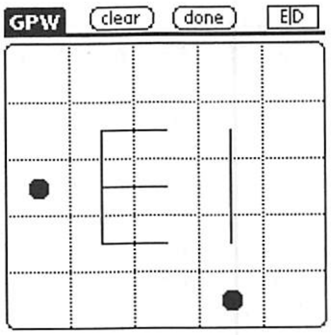
Due to the dependence of the security of a scheme on the passwords that users choose in practice, a new password scheme can not be *proven* better than an old scheme. Performing trials on subjects in order to learn the distribution of user passwords for a new scheme is impractical for such large sample spaces. In the case of textual passwords, learning the knowledge that attackers routinely use would correspond to trying to learn the English dictionary (among others) given no prior knowledge of the types of letter combinations used in English, by having subjects type in 8-character passwords. In the absence of such objective proof, we present three plausibility arguments that suggest that the DAS scheme is considerably harder to crack than the conventional textual scheme. Two of these are estimates of the information content of the DAS password space, which we argue improves on the information content available with textual passwords. The third argument discusses the effect that lack of knowledge of the distribution of user choices has on an attacker.

3.3.1 The Size of the Password Space

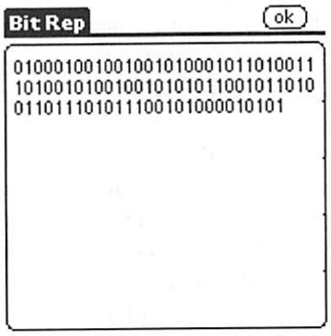
First we consider the raw size of the password space, or in other words, its information content assuming users are equally likely to pick any element as their password. The raw size is an upper bound on the information content of the distribution that users choose in practice. We need some way to delimit the password space in order to obtain a finite answer, or in probabilistic terms, a way to ascribe probability zero to an infinite subset of passwords, leaving a finite subset that we will count. We assume that all passwords of total length (as defined in Section 3.1) greater than some fixed value have probability zero. We compute the size $\Pi(L_{\max}, G)$ of the space of passwords of total length less than or equal to $L_{\max}$ on a grid of size $G \times G$. Π is defined in terms of the



(a) User inputs desired secret

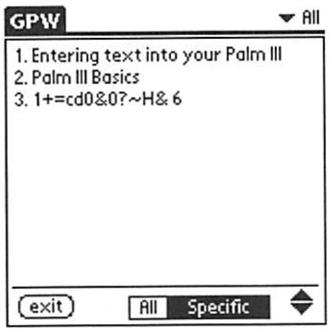


(b) Internal representation



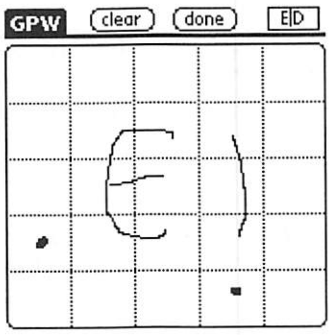
```
0100010010010010100010110100 11
1010010100100101010110010110 10
0110111010111001010000 10101
```

(c) Raw bit string

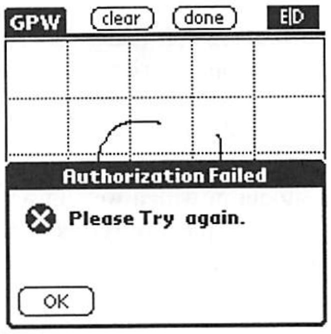


1. Entering text into your Palm III
2. Palm III Basics
3. 1+=cd0&0?~H& 6

(d) Interface to database



(e) Re-entry of (incorrect) secret



Authorization Failed

Please Try again.

(f) Authorization failed

Figure 3: A password is created by drawing the secret on the display as shown in (a). Both the internal representation of the input password showing the cells covered by the user's drawing and the derived key are depicted in (b) and (c) respectively. To apply a symmetric cryptographic function to records in the database (shown in (d)), the user selects the records and then re-inputs the DAS password. If the encryption of a known cleartext with the input password matches the stored ciphertext created during initialization, then the symmetric cryptographic routine, $E_k(x)$, is applied to the selected records. Otherwise, the user is prompted to re-enter the DAS secret.

number of passwords with total length equal to L, $P(L,G)$ by:

$$\Pi(L_{\max},G) = \sum_{L=1}^{L_{\max}} P(L,G)$$

In turn, $P(L,G)$ can be defined in terms of $N(l,G)$, the number of strokes of length equal to l by:

$$P(L,G) = \sum_{l=1}^{l=L} P(L-l,G)N(l,G)$$

That is, a new stroke of length l may be added to any shorter password of length $L-l$ to make a password of total length L. By defining $P(0,G)=1$, we complete the definition of the recurrence, once we have given an expression for $N(l,G)$.

The following recurrence relation defines $N(l,G)$. Let $n(x,y,l,G)$ be the number of strokes of length l ending at the cell (x,y) in a grid of size $G \times G$. Then N can be defined in terms of n by

$$N(l,G) = \sum_{(x,y)\in[1..G]\times[1..G]} n(x,y,l,G)$$

Clearly, $\forall(x,y) \in [1..G]\times[1..G], n(x,y,1,G) = 1$. Moreover, it is convenient to define $\forall(x,y) \notin [1..G]\times[1..G], n(x,y,l,G) = 0$. The function n can then be evaluated using the following recurrence:

$$\begin{aligned} n(x,y,l,G) &= n(x-1,y,l-1,G) \\ &+ n(x+1,y,l-1,G) \\ &+ n(x,y-1,l-1,G) \\ &+ n(x,y+1,l-1,G) \end{aligned}$$

Putting the pieces together, we can calculate the size of the password space. The results for different upper bounds on total password length on a 5×5 grid are given in Table 1.

The data in Table 1 shows that the raw size of the graphical password space surpasses that of textual passwords for reasonable password configurations. While these numbers are encouraging, in practice not all graphical passwords are equally likely to be chosen by users, rendering a uniform distribution overly optimistic. For example, although the number of passwords of length greater than or equal to 12 is already greater than the number of textual passwords of 8 characters or less constructed from the printable ASCII codes ($95^8 \approx 2^{53}$), this includes all possible combinations of twelve isolated dots.

In order to obtain a more realistic estimate of the information content, in the following section we suggest a model in which we characterize passwords as being "memorable" in terms of the programs which generate them.

3.3.2 Modeling User Choice

We assume that the reason that users choose from such a small subset of textual passwords is that the passwords in that set are more memorable than those outside it. That lack of imagination on the part of the user is not the cause for the lack of variety is supported by the fact that system-generated passwords have been so unsuccessful [2]. By making the same assumption about DAS passwords, we can "reduce" our task to that of modeling the set of "memorable" graphical passwords. If we can show that this set, or some subset of it, has cardinality larger than the dictionary of textual passwords from which users typically choose, we can plausibly claim that as far as information content goes, DAS is more secure than conventional textual password schemes. Here, we identify two such subsets using different criteria of memorability, and show that the cardinalities of these sets do indeed satisfy the above criterion.

What constitutes a memorable password? In the textual case, one obvious component is semantic content. If the sequence of characters has a meaning for the user, the password is more likely to be memorable [18, 27, 6]. This semantic definition is extremely hard, if not impossible, to characterize in the abstract. It is only because the semantic content of many character combinations has been established by the common use of a written language that we can talk about such content at all. In the DAS scheme, there are obvious password components that have meaning, but it is impossible *a priori* to identify exactly which passwords will have semantic content, and to how many users, precisely because it is not a representation with meanings established by common use.

Memorability based on simple shapes The first set of "memorable" passwords that we define is a subset of those passwords that might reasonably be expected to carry meaning. We look at all strokes in the form of rectangles, and show that by combining two such strokes, we already reach the size of the dictionaries used to crack textual schemes. To be more precise, consider the set of rectangles within a $G \times G$ grid. Since a rectangle can be defined by two rows (the top and bottom edges of the rectangle) and two columns (the left and right edges), it is clear that the number $R(G)$ of rectangles on a $G \times G$ grid is

$$R(G) = \binom{G}{2}^2 = \frac{1}{4}G^2(G-1)^2$$

Each of these rectangles can be generated in many ways. For example, the starting point of a stroke can be at any of the corners, and the stroke direction can be clockwise or counter-clockwise. This yields 8 possibilities for each rectangle. In addition, one can choose whether to close the rectangle by returning to the starting cell or not, again doubling the possibilities. On a 5×5 grid, this amounts to 1600 possible strokes. Two such strokes in succession gives 2.56×10^6 passwords, already roughly the size of the textual dictionary that contained the passwords of 25% of users in Klein's study [12]. Clearly we can generate a much larger set of passwords by considering variations on the theme of rectangles, or by considering other Gestalt forms [33].

Memorability based on short algorithms The second set of passwords that we describe is suggested by the discussion of text-based graphical passwords in Section 2, which pointed toward a different definition of memorability. There, a memorable sequence of positions seemed characterized by the fact that there existed a short algorithm to

$L_{\max}$	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
$\log_2$(# passwords)	5	10	14	19	24	29	33	38	43	48
$L_{\max}$	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
$\log_2$(# passwords)	53	58	63	67	72	77	82	87	91	96

Table 1: Number of passwords of total length less than or equal to $L_{\max}$ on a 5×5 grid.

describe the sequence. It is this definition of memorable that we wish to apply here, since it can be characterized in precise terms. We do not argue that every memorable password has a short algorithm to describe it, but that passwords describable by short algorithms are memorable. We will show that the cardinality of this subset of memorable passwords is already larger than the dictionary of character sequences from which users most often draw their passwords, and that therefore, following the argument above, the DAS password scheme should be harder to crack in practice than the conventional textual scheme.

In order to characterize the "complexity" of the algorithm required to generate a DAS password, we define a very simple language suited to the task of describing DAS passwords. Then, we generate all programs in this language whose complexity is at most a chosen maximum. In order to avoid counting different programs that produce the same password twice, we then execute the generated programs to output the passwords, which are then bucketed, and distinct passwords counted. The result is the number of DAS passwords generated by programs of complexity at most the chosen maximum.

Before describing the results of this endeavor, we give some details of the language in which we generated the programs. The grammar of the language is as follows:[2]

```
  program  →  digit digit block
    block  →  statement block
statement  →  instr |
              repeat digit block end
    instr  →  up | down | right |
              left | penup | pendown
    digit  →  1 | 2 | 3 | 4 | 5
```

[2]Those readers old enough to remember the APPLE II will recognize that our language bears a striking resemblance to LOGO [31].

The first two digits represent a starting position. The instructions **up**, **down**, **left**, and **right** move the pen one square in the indicated direction. If the pen is currently in the down position, then moving in the specific direction will draw a line. Otherwise, the direction statement will merely move the pen location. The pen begins in the up position. The **repeat** statement is our iterator. We allow digit values up to the number of grid squares on each axis (i.e., 5 on a 5×5 grid) to indicate the number of repetitions, although in principle a password consisting of more than 5 repetitions of something on a 5×5 grid are possible (e.g., ten dots in the same position).

To calculate the complexity for a given program, we assign a complexity to each literal in our language. We assign every statement and digit complexity one, except for the **end** marker, which has complexity zero. This means that **repeat** loops have a complexity of two (one for the **repeat** statement, and one for the integer indicating the number of repetitions) plus the complexity of the repeated block. In addition, the last **penup** statement of a program is assigned a complexity of zero (lifting one's pen from the surface at the end of entering a password is difficult to forget). So, for example, there are no programs of complexity only two, since the integers describing the starting position of the program already consume a complexity of two without allowing any **pendown** statements. The first complexity of which there are any programs is three—the two digits describing the initial starting position, followed by a **pendown**—and the passwords generated by programs of complexity three are simply those consisting of a single tap on one of the grid squares. Note that our complexity calculations for programs are very conservative, in the sense that even pen movements *between* strokes (i.e., while the pen is raised) are counted in the complexity of a program.

The results of using the above described procedure for counting the number of DAS passwords of a given complexity on a 5×5 grid are shown in Figure 4. As expected, this data shows that the number of DAS passwords grows exponentially as a function of the maximum complexity of the program. What is more interesting, however, is that by extrapolation[3] we see that the number of DAS passwords generated by programs of only complexity 12 far surpasses the dictionary size of approximately 3×10^6 used in Klein's password-cracking studies [12]. As a point of comparison, even just tracing the outermost cells of a 5×5 grid to make a square already requires a program of complexity at least fifteen in our simple language. And, obviously this design and many other, more complex ones will fall in the realm of memorable for most users. We believe that this is compelling evidence that DAS passwords, of which those generated by programs of complexity at most twelve are but a very small subset, will be significantly harder to crack in practice than textual passwords. Example DAS passwords and the shortest programs that generate them are given in Appendix B.

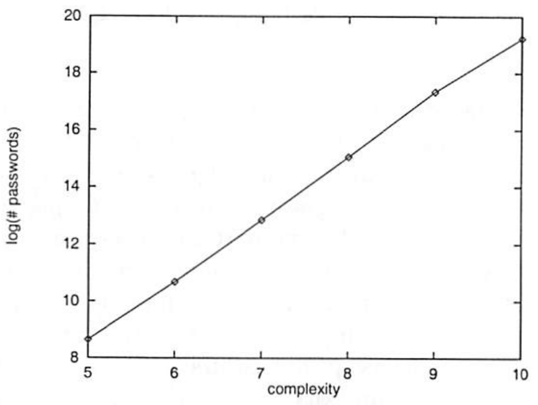


Figure 4: Number of DAS passwords generated by programs of small complexity on a 5×5 grid.

3.3.3 Lack of Knowledge of the Distribution

Given the size of typical password spaces, knowledge of the distribution of user passwords is essential to an adversary. Without such knowledge the adversary has no way of directing her search toward more probable passwords, and is no better off than if users really did pick their passwords uniformly from the set of possibilities [8].

[3]Calculating the exact number of "memorable" graphical passwords, as defined by our language, for complexities greater than 10 requires significantly more computational resources (and time) than we have available to us. An attacker wishing to build any such database will face similar difficulties.

Where did the knowledge of the distribution come from in the case of textual passwords? For the most part, dictionaries have been compiled by using reasonable assumptions about likely choices. The assumptions stem from the use of a shared language, and shared knowledge of the semantic content of words. For example, in the work of Klein [12] the sources for likely passwords included the St. James Bible, the `Unix` dictionary, and many other sources of English words that were available to the author precisely because they are a part of our language. If these assumptions had turned out to be incorrect, textual password schemes would be extremely difficult to break in practice.

The assumptions made about likely password choices are strongly confirmed by Klein's work, and by successful attacks on textual passwords, but confirmation of pre-existing dictionaries is not the same as deriving a dictionary in the first place by learning from example without prior knowledge. In the case of textual passwords, this would mean learning the English dictionary (or some equivalent corpus of words) by collecting user passwords. This would involve acquiring millions of verified passwords, and, as such, represents a significant challenge for a would-be adversary.

In the case of the DAS scheme, similar reasonable assumptions about user choice do not exist. Furthermore, the learning task is made even more difficult by two factors. First, our previous arguments suggest that both the space of passwords and the space of likely user choices are considerably larger than for textual passwords. Second, the platform that we are targeting, PDAs, renders the task of data collection much harder than on, e.g., networked computers.

4 Prior Work

There is a considerable amount of prior work on authenticating users via graphical inputs to a device, particularly handwritten signatures (see, e.g., [14, 13, 21]). None of these works strive for exact repeatability by the user, and therefore, a model of the user's graphical input is stored on the device and used to ascertain whether a new input is sufficiently similar to the previously-stored one to grant access. This renders it essential to protect the device's (PDA's) storage from probes by an attacker.

In contrast, repeatability is achieved in our schemes, thereby enabling designs in which the device, if captured, is of little help to the attacker (see Section 1).

The security of textual passwords has been examined by numerous researchers, notably [19, 12, 9, 29, 34]. Without exception, these studies reiterate the fact that people choose passwords that are easy to find by automated search. In order to improve the security of passwords, it is common practice for system administrators to invoke reactive password checkers to identify weak passwords [26, 20], or to use proactive checkers to filter out certain classes of weak passwords when the user inputs her password for the first time [3, 28].

A technique to improve the security of even a poorly chosen password is to *salt* the password by prepending it with a random number, R, before hashing [19, 16]. The effect is that the search space of the attacker is increased by a factor of $2^{|R|}$ if the attacker does not have access to the salts.

The techniques in this paper can be combined in natural ways with the techniques discussed above for strengthening textual passwords—i.e., proactive and reactive password checking, and salting—to improve the security of graphical passwords, as well.

More distantly related is work on one-time passwords (e.g., [11]). One-time password schemes are relevant primarily for network settings, to defend against the threat of a network eavesdropper capturing password information in transit between the user and a secure authentication server. To render such eavesdropping harmless, a one-time password scheme varies the user's password from each login to the next in a way that only the user and the server can predict, based on state shared between the server and user. In the main setting we consider, however, there is no network communication that is vulnerable to eavesdropping, and consequently the attacks with which we are concerned is the capture and analysis of all stored state relevant to authentication (the PDA in our setting, or equivalently the server's and client's states in the one-time password setting). One-time password schemes of which we are aware offer no benefit against this attacker over traditional password schemes.

5 Conclusions and Future Work

We have presented graphical password schemes that achieve better security than conventional textual passwords. Our approaches exploit the input capabilities of graphical devices that allow us to decouple the position of inputs from the temporal order in which they occur. We presented arguments for the security of our schemes in which we analyzed the information content of the resulting password spaces. We also presented a novel approach for capturing the "memorability" of graphical passwords by examining the class of DAS passwords generated by short programs in a simple grid-based language, and showed that even this relatively small subset of graphical passwords (for some fixed program complexity) constitutes a much larger password space than the dictionaries of textual passwords to which a high percentage of passwords typically belong.

For future work we are exploring alternative schemes for modeling the memorability of DAS passwords that we hope will capture their high-level structure more intuitively than our current models. The goal is to capture the concept of organized drawings, in which the view of the whole is more than just the sum of the individual parts that constitute it. For example, one can view a square as an object in itself and not simply as an arrangement of the individual lines from which it is composed. In this way, we can define a set of primitive structures from which all "memorable" drawings can be derived using meta-level compositions of these primitives. We hope to show that even this reduced set of DAS passwords (for some reasonable number of primitives) constitutes a larger space than that of textual-based passwords, and as such will be significantly harder to crack in practice.

6 Acknowledgement

This work is partially sponsored by a USENIX Scholars Research Grant; and by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-96-1-0320 and F30602-99-1-0517; and by the National Science Foundation under grant number CCR-9411590. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright anno-

tation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Laboratory, or the U.S. Government.

References

[1] A. Alvare. How crackers crack passwords or what passwords to avoid. In *Proceedings of the* 2nd *USENIX Security Workshop*, August 1990.

[2] M. Bishop. Password management. In *Proceedings of COMPCON '91*, pages 167–169, February 1991.

[3] M. Bishop. Improving system security via proactive password checking. *Computers and Security*, 14(3):233-249, April 1995.

[4] G. Blonder. *Graphical passwords.* United States Patent 5559961, 1996.

[5] G. H. Bower, M. B. Karlin, and A. Dueck. Comprehension and memory for pictures. *Memory and Cognition*, 2:216–220, 1975.

[6] M. A. Borges, M. A. Stepnowsky, and L. H. Holt. Recall and recognition of words and pictures by adults and children. *Bulletin of the Psychonomic Society*, 9:113–114, 1977.

[7] M. W. Calkins. Short studies in memory and association from the Wellesley College Laboratory. *Psychological Review*, 5:451–462, 1898.

[8] T. M. Cover,and J. A. Thomas. *Elements of Information Theory*, John Wiley and Sons, 1991.

[9] D. Feldmeier and P. Karn. UNIX password security—Ten years later. In *Advances in Cryptology—CRYPTO '89 Proceedings* (Lecture Notes in Computer Science 435), 1990.

[10] S. Garfinkel and E. Spafford. *Practical Unix & Internet Security.* O'Reilly & Associates, Inc., 1996.

[11] N. Haller. The s/key(tm) one-time password system. In *Proceedings of the 1994 Symposium on Network and Distributed System Security*, pages 151–157, February 1994.

[12] D. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *Proceedings of the* 2nd *USENIX Security Workshop*, August 1990.

[13] F. Leclerc and R. Plamondon. Automatic signature verification: The state of the art—1989–1993. *International Journal on Pattern Recognition and Artificial Intelligence*, 8(3):643–660, June 1994.

[14] G. Lorette and R. Plamondon. Dynamic approaches to handwritten signature verification. In *Computer Processing of Handwriting*, pages 21–47, World Scientific, 1990.

[15] S. Madigan. Picture memory. In *Imagery, Memory, and Cognition*, pages 65–86, Lawrence Erlbaum Associates, 1983.

[16] U. Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security*, 15(2):171–176, 1996.

[17] G. Mandler. Your face looks familiar but I can't remember your name: A review of dual process theory. *Relating Theory and Data*, 207–225, 1991.

[18] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.

[19] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, November 1979.

[20] A. Muffet. Crack: A sensible password checker for Unix. Available via anonymous *ftp* from *cert.org.*

[21] V. S. Nalwa. Automatic on-line signature verification. *Proceedings of the IEEE*, pages 215–239, February 1997.

[22] D. L. Nelson, U. S. Reed, and J. R. Walling. Picture superiority effect. *Journal of Experimental Psychology: Human Learning and Memory*, 3:485–497, 1977.

[23] A. Paivio. *Imagery and Verbal Processes.* Holt, Rinehard, and Winston, New York, 1971.

[24] A. Paivio. Imagery in recall and recognition. *Recall and Recognition*, John Wiley, New York, 1976.

[25] A. Paivio, T. B. Rogers, and P. C. Smythe. Why are pictures easier to recall than words? *Psychonomic Science*, 11:137–138, 1968.

[26] T. Raleigh and R. Underwood. CRACK: A distributed password advisor. In *Proceedings of the* 1st *USENIX Security Workshop*, pages 12–13, August, 1988.

[27] R. N. Shepard. Recognition memory for words, sentences, and pictures. *Journal of Verbal Learnings and Verbal Behavior*, 6: 156–163, 1967.

[28] E. Spafford. Preventing weak password choices. In *Proceedings of the* 14th *National Computer Security Conference*, pages 446–455, October 1991.

[29] E. Spafford. Observations on reusable password choices. In *Proceedings of the* 3rd *USENIX Security Symposium*, September 1992.

[30] L. Standing. Learning 10,000 pictures. *Quarterly Journal of Experimental Psychology*, 25:207–222, 1973.

[31] Cynthia J. Solomon and Seymour Papert. A case study of a young child doing Turtle Graphics in LOGO. *MIT AI memo 375*, July 1976.

[32] J. E. Wells. Encoding and memory for verbal and pictorial stimuli. *Journal of Experimental Psychology*, 24:242–252, 1972.

[33] Max Wertheimer. *Laws of organization in perceptual forms.* A source book of Gestalt psychology (pp. 71-88). London: Routledge & Kegan Paul. 1938.

[34] T. Wu. A real-world analysis of Kerberos password security. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, 1999.

A A Picture is Worth a Thousand Words

Our "draw-a-secret" scheme is motivated by the experimentally-proven fact that pictures are easier to remember than words. Why are pictures easier to recall? Four hypotheses have been offered as explanations of picture-word differences in recall:

- Common-code theory: this view of memory and recall theorizes that pictures and words access semantic information in a single conceptual system that is neither word-like or picture-like. This theory hypothesizes that pictures and words both require analogous processing before accessing semantic information, but pictures require less time than words for accessing the common conceptual system. Common-code theorists attribute better picture recall to differences in the encoding of pictures and words: pictures share fewer common perceptual features among themselves and therefore need to be discriminated from a smaller set of possible alternatives than words. The greater number of dictionary meanings or the greater lexical complexity of words create uncertainty and confusion, and hence poorer recall.

- Dual-code theory: unlike the common-code approach, this theory postulates that language and knowledge of worlds are represented in functionally distinct verbal and nonverbal memory systems. The verbal system is specialized for dealing with linguistic information whereas the non-verbal stores perceptual information. The most evident examples of dual process theory can be found in experiences that we have all had at some time or the other: we meet someone, know them to be familiar but do not know who they are; we recognize a melody, but fail to remember its name or when or where we heard it before; we read a line of a poem, know it, but do not know where we have read it before, much less the title or author of the poem. In all these cases, we experience a sense of familiarity, but have — at least at first — no access to any contextual or conceptual information [17].

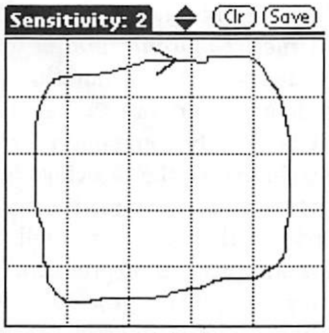
  Dual code theory suggests that there are qualitative differences between the ways words and pictures are processed during memory and hypothesizes that the reason for superior picture memory is that pictures automatically engage multiple representations and associations with other knowledge about the world, thus encouraging a more elaborate encoding than occurs with words [25, 23].

- Abstract-propositional theory: in contrast to the dual-code approach, this theory rejects any notion of sophisticated distinctions between verbal and nonverbal modes of representation, but instead describes representations of experience or knowledge in terms of an abstract set of relations and states, called propositions. This theory postulates that better free recall with pictures may be due to even more elaborative encoding effects than those suggested by dual-code theorists. Propositional theorists view the involvement of *abstractive and interpretive* processes in picture memory as the explanation for the picture effect [15]. Therefore, a series of line drawings will be poorly remembered if a subject is unable to interpret the drawings in a meaningful way, whereas memory for the same drawings, presented in the same way will be much better if a conceptual interpretation is provided, and it this interpretive process which is responsible for better picture memory recall.

While the strongest evidence thus far for the picture effect can be best explained by dual-code theory (see [17]), an understanding of picture memory and the means by which we acquire and maintain information about the visual environment is still an ongoing challenge. Nonetheless, the research to date provides strong arguments in terms of the memorability of drawings over words in recognition tasks and hence its applicability to computer security.

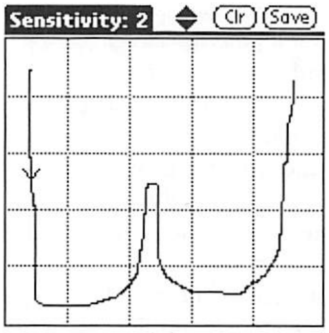
B Example DAS Passwords

The examples illustrated in Figure 5 show a few DAS passwords along with the shortest programs which generate them (using the grammar outlined in Section 3.3.2) and their respective complexities.



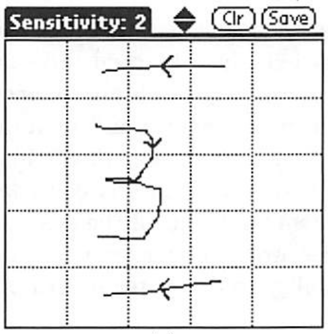
```
1 1
pendown
repeat 4
right
end
repeat 4
down
end
repeat 4
left
end
repeat 4
up
end
penup
```

(a)



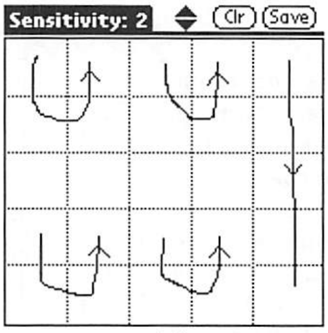
```
1 1
pendown
repeat 4
down
end
right
right
up
up
down
down
right
right
repeat 4
up
end
penup
```

(b)



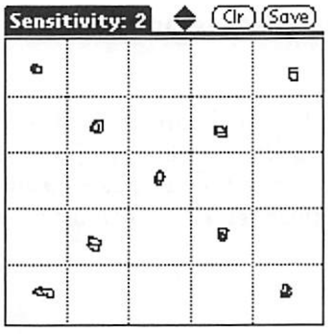
```
2 2         end
pendown     right
repeat 2    right
right       pendown
down        left
left        left
end         penup
penup
right
right
down
pendown
left
left
penup
repeat 4
up
```

(c)



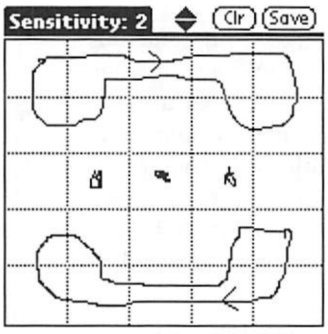
```
1 1         repeat 3
repeat 2    up
pendown     end
down        right
right       end
up          pendown
penup       repeat 4
left        down
repeat 3    end
down        penup
end
pendown
down
right
up
penup
```

(d)



```
1 1         repeat 2
pendown     penup
penup       down
repeat 4    left
right       pendown
end         end
pendown     penup
penup       right
down        right
repeat 3    pendown
left        penup
end         down
pendown     repeat 3
penup       left
right       end
right       pendown
pendown     penup
```

(e)



```
1 1         end         penup
pendown     repeat 4    up
repeat 4    right       repeat 4
right       end         left
end         pendown     end
down        down        repeat 3
left        repeat 4    left
up          left        pendown
left        end         penup
left        up          end
down        right
left        down
up          left
penup       left
repeat 3    up
down        right
```

(f)

Figure 5: The drawings above have complexities 15, 17, 24, 26, 39, and 42, respectively (recall that final pen-ups have zero cost).

Hand-Held Computers Can Be Better Smart Cards

Dirk Balfanz
Princeton University
balfanz@cs.princeton.edu

Edward W. Felten
Princeton University
felten@cs.princeton.edu

Abstract

Smart cards are convenient and secure. They protect sensitive information (e.g., private keys) from malicious applications. However, they do not protect the owner from abuse of the smart card: An application could for example cause a smart card to digitally sign any message, at any time, without the knowledge of the owner.

In this paper we suggest that small, hand-held computers can be used instead of smart cards. They can communicate with the user directly and therefore do not exhibit the above mentioned problem.

We have implemented smart card functionality for a 3COM PalmPilot. Our implementation is a PKCS#11 module that plugs into Netscape Communicator and takes about 5 seconds to sign an email message.

Generalizing from this experience, we argue that applications that are split between a PC and a hand-held device can be more secure. While such an application remains fast and convenient to use, it gains additional security assurances from the fact that part of it runs on a trusted device.

1 Introduction

Public key systems (like RSA [Rivest et al., 1978] or DSA [DSA, 1994]) promise to play a major role in the evolving global networked community. Since they solve the key distribution problem, they are especially useful for a big, open network like the Internet. In a public key system every individual possesses two keys: a *public* and a *private* key. The public key is published and known to everyone. The private key is only known to its owner, and kept secret from everyone else. Public keys are used to encrypt messages and verify signatures, and private keys are used to decrypt messages and produce signatures. If a thief steals your private key, he can not only read confidential messages sent to you, he can also impersonate you in electronic conversations such as email, World Wide Web connections, electronic commerce transactions, etc.

People should therefore protect their private keys adequately. One way to do so are *smart cards*. Smart cards are small[1], tamper-resistant devices that can be connected to a PC and store private keys. The PC cannot learn the private key stored on the smart card, it can only ask the card to perform certain cryptographic functions for which the private key is needed, such as calculating a digital signature or decrypting an email message. These functions are executed on the smart card. This way, the connected PC can be used to engage in electronic conversations on behalf of the smart card owner, but it does so without knowledge of the owner's private key.

Contrast this with a scenario in which the private key is stored on the PC itself (which is the most common case today). Although the key will probably be protected by a password, the PC itself could fall victim to an attack from the outside ([Gutmann, 1997]). If and when that happens, it is usually just a question of determination for the intruder to break the protection mechanism and learn the private key. Even worse, if the application that uses the private key turns out to be malicious (it could be infected by a virus, or turn out to be a tampered-with version of a well-known application, or some other Trojan horse), it can get to know the private key without any further ado.

[1] Usually, smart cards have the form factor of a credit card.

Smart cards protect users from security breaches of that kind. They are also quite convenient. Since they allow users to carry around their private keys with them, and at the same time connect to any PC, they allow users to engage in electronic conversations from any PC.

However, smart cards also have problems. Imagine, like in the scenario described above, that the PC to which the smart card is connected has been compromised, and that an email application that talks to the smart card is malicious. While the application will not be able to learn the private key on the smart card, it may do other things that are almost as bad. For example, when the user composes an email message to `president@whitehouse.gov` that reads

> I support you,

then the malicious emailer could, instead, send out an email that reads

> I conspired to kill you,

and could even have the smart card sign the latter. The digital signature might stand up in court and could get the owner of the private key into a lot of trouble.

The problem here is that the smart card does not have an interface to the user. A smart card is designed to protect the private key even if it is talking to a malicious application. However, it does not protect against *abuse* of the private key as shown in the example above. If the smart card had a user interface it could have shown – for verification purposes – the message it was asked to sign, and the user would have learned that the emailer was trying to frame him.

Recently, hand-held computers like the 3COM PalmPilot emerged in the marketplace. They are almost as small as smart cards, have superior computing power[2], and provide an interface to the user. In this paper we report on how we implemented smart card functionality for the 3COM PalmPilot (which we will simply call "Pilot" from now on). We describe how even in our initial attempt we achieved certain security properties that normal smart cards cannot deliver.

In Section 2 we give an overview of PKCS#11, which is a standard for "cryptographic tokens" (smart cards). In Section 3 we describe details of our implementation. In Section 4 we give an estimate as to how hard or easy it would be to extract the private key from a Pilot. Section 5 is reserved for a outlook into the future: What else can be done with hand-held computers? We summarize and conclude the paper in Section 6.

[2]With the exception of certain cryptographic operations - see Section 3.5.

2 Overview of PKCS#11

We have implemented a PKCS#11 library for Netscape Communicator. PKCS#11 [PKCS#11, 1997] is a "standard" drafted by RSA Data Security Inc. It defines sixty-odd prototypes for functions that together can be used to perform a wide range of cryptographic mechanisms, including digital signatures, public key ciphers, symmetric key ciphers, hash functions etc. The standard is designed with smart cards in mind[3]: The caller can have the PKCS#11 library perform certain functions, like producing a signature, without ever knowing the private key that is used in the process. In this section we will explain how PKCS#11 works, how Netscape Communicator uses it, and what the design of our PalmPilot implementation is.

PKCS#11 describes function prototypes and semantics for a *library*, which we will call *PKCS#11* or *cryptoki library*. An application can bind to a cryptoki library and call into its functions to perform certain cryptographic operations. The PKCS#11 world is populated by *objects*, which are sequences of attribute-value pairs. For example, the following is an object:

Attribute	**Value**
OBJECT_CLASS	PRIVATE_KEY
KEY_TYPE	RSA
TOKEN	YES
EXTRACTABLE	NO
LABEL	BOBS_PRIVATE_KEY
PRIVATE_EXPONENT	8124564...
MODULUS	7234035054...
⋮	⋮

Figure 1: A private RSA key

There are five different classes of objects: certificates, public keys, private keys, secret (i.e., symmetric) keys, and "data" objects. Objects can be created temporarily (e.g., a session key for an SSL connection), or can be

[3]It is also called the *cryptoki*, or cryptographic token interface, standard.

long-lived, in which case they are assumed to be stored on a "cryptographic token". The example above describes a private RSA key. It is stored on a smart card and marked "unextractable". This means that calls into the cryptoki library that try to read the values of various attributes of this key will fail. Hence, there is no way for the application to learn the value of this secret key (assuming that in fact is is stored on a smart card and not in the same address space as the application).

How can the application use this key to sign a message? The cryptoki library returns *handles* to the application, which are usually small integers and uniquely identify objects. Although the application will not be able to learn the private exponent of the key above, it can for example ask the cryptoki library if it knows of an object with the label "BOBS_PRIVATE_KEY". If such an object is found, a handle H is returned to the application, which can then ask the cryptoki library to sign a certain message with the object identified by H. Using the information known to it, the cryptoki library can sign the message and return the signature to the application. If the object is stored on the smart card, and the handle, message and signature are the only things exchanged between smart card and application, then the private key is safe. Object search and signature calculation have to happen on the smart card in this case.

PKCS#11 defines many more functions besides object search and message signing. The application can learn which cryptographic mechanisms are supported by the library, create and destroy objects, encrypt and decrypt messages, create keys, and more.

The cryptoki library will not perform certain functions (like signing a message) unless the user is *logged on* to the cryptographic token. Usually, the user has to provide a PIN or password to log on to the token. PKCS#11 distinguishes between tokens with or without a *trusted authentication path*. A token with a trusted authentication path is a smart card to which the user can log on directly, e.g., by means of a numeric key pad on the card. If the card does not have a trusted authentication path, then the application on the PC has to gather the PIN or password from the user, and send it to the smart card. We note that this is less secure than a trusted authentication path: The application can learn the PIN or password needed to "unlock" the private key on the smart card. Smart cards usually do not have a trusted authentication path.

Netscape Communicator supports PKCS#11, which means that smart card vendors can provide a cryptoki shared library [PKCS#11, 1998]. Communicator will then bind to that library and use it for certain cryptographic operations. In particular, Communicator uses the library for key pair generation, S/MIME encrypted email [Dusse et al., 1998a, Dusse et al., 1998b] and client authentication in SSL connections [Freier et al., 1996]. For S/MIME the cryptoki library has to support RSA [Rivest et al., 1978], DES [DES, 1993], RC2 [Rivest, 1998], and Triple-DES [Schneier, 1996]. Communicator only supports RSA key pair generation, so client authentication – although technically feasible with any signature scheme – is done using RSA signatures.

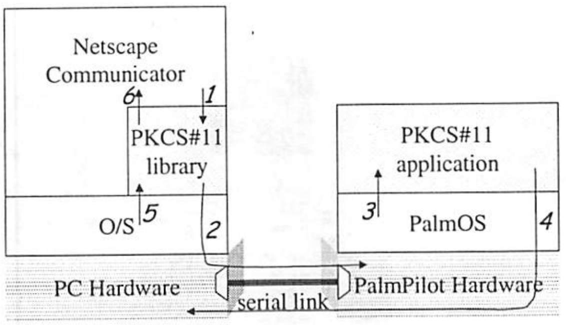


Figure 2: Overview of our PKCS#11 implementation

We implemented a cryptoki library that supports the above mentioned ciphers and signature schemes. The shared library that plugs into Communicator only serves as a dispatcher of requests to the Pilot. For the Pilot, we have written an application that receives those requests, performs the necessary functions and sends the results back to the PC.

Figure 2 shows how our implementation works. The PC and the PalmPilot are connected through a serial link. We implemented the pieces shown in grey: a plug-in library for Netscape Communicator and an application for the PalmPilot. Communicator calls into our library when some cryptographic operation needs to be performed (1). If the request cannot be handled by the library itself, it forwards it to the PalmPilot (2). On the PalmPilot, the operating system notifies our PKCS#11 application when a request arrives (3). The application processes the request and returns the result to the PC (4), where it is received by the cryptoki library (5). The library will then return this result or a result based on what it received from the PalmPilot back to Communicator (6).

It is worth pointing out what the trusted components are in this model: We trust that the PKCS#11 application on the PalmPilot is not tampered with and performs correctly. We trust in the same way the operating system on the PalmPilot and its hardware. On the other hand, we

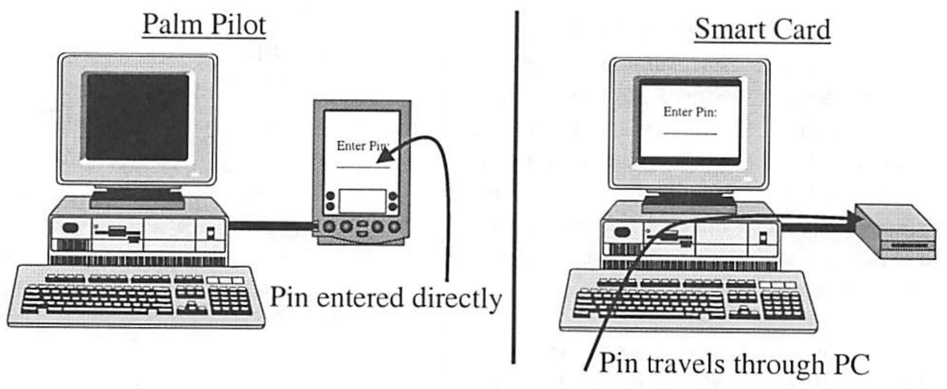


Figure 3: Information flow for PIN, contrasting traditional smart card and PalmPilot.

do not have to trust Communicator, the operating system on the PC or its hardware to collaborate with us. We do not even trust that the PKCS#11 library does what it is supposed to do. The PKCS#11 application on the Pilot is written defensively and works[4] even in the case where the PKCS#11 library is replaced by malicious code.

3 The Implementation

Our implementation runs on Windows 95, Windows NT, and Linux for the Communicator plug-in, and on a 3COM PalmPilot Professional for the Pilot side. It practically turns the Pilot into a smart card. For the cryptographic functions we used the PalmPilot port of SSLeay [Young et al., 1998]. In the following section we will highlight a few points of our particular implementation.

3.1 Key Pair Generation

To create an RSA key pair, we need a good source of randomness. In a traditional smartcard, if the attacker knows the random number generating algorithm and initial seed, we cannot hide the private key from him. He can simply perform the same computation as the smart card. PKCS#11 provides functions to re-seed the random number generator on the card, but the application never has to call them.

On the Pilot, we can use external events to provide randomness that the application cannot observe. When the Pilot generates a new key pair, we ask the user to scribble randomly on the screen of the Pilot, thus providing random bits we use for seeding the random number generator.

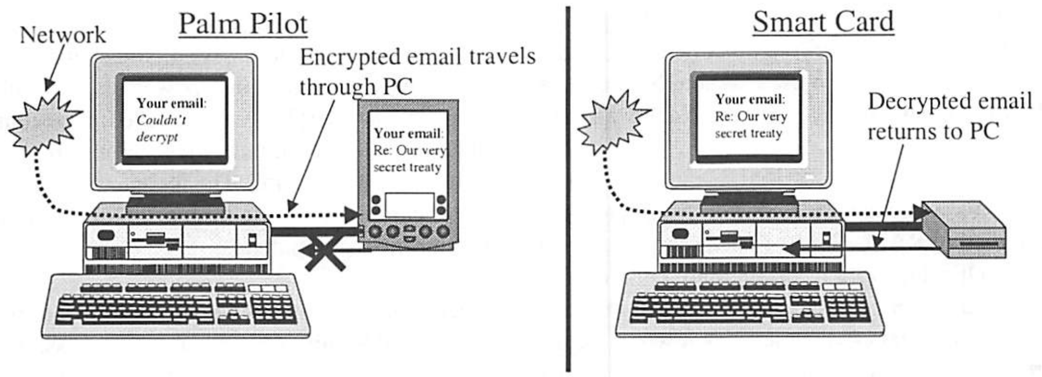
[4]"Works" means that it does not leak any information it is not supposed to leak. It does not mean that the system does not crash, for example.

3.2 Logging on to the Pilot

The Pilot is a cryptographic token with a trusted authentication path. Therefore, the user does not enter his password through Communicator, but directly into the Pilot. This way Communicator cannot learn the PIN or password needed to unlock the private keys. If it knew the PIN or password, it could try to use the Pilot without the user's knowledge or - even worse - make the PIN known to adversaries who might later find an opportunity to steal the Pilot from its owner. Figure 3 shows the difference in information flow between a traditional smart card and our PalmPilot implementation.

3.3 Signing Email Messages

In order to send a signed email message, all the user needs to do is log on to the Pilot and put it into its cradle, which will connect it to Communicator. Communicator will ask the Pilot to sign the message and also retrieve a certificate stored on the Pilot (unlike private keys, certificates are extractable objects and can therefore be retrieved by the application). Then, Communicator adds the signature and the certificate to the message according to the S/MIME standard, and sends it off.



Palm Pilot
Network
Encrypted email travels through PC
Your email: Couldn't decrypt
Your email: Re: Our very secret treaty
Smart Card
Your email: Re: Our very secret treaty
Decrypted email returns to PC

Figure 4: Information flow for reading encrypted email, contrasting traditional smart card and PalmPilot.

3.4 Receiving Encrypted Email

When Communicator receives a message that has been encrypted with the user's public key, it will first ask the Pilot to use the corresponding private key to unwrap the symmetric (RC2 or DES) key used to encrypt the contents of the message. Depending on the preference setting on the Pilot, it will either signal an error, or unwrap the symmetric key and store it as an unextractable object on the Pilot. In the former case, Communicator will ask to "decrypt" a certain string which happens to be the wrapped key, and hence obtain the symmetric key needed to decrypt the message.[5] In the latter case, Communicator will send the encrypted message to the Pilot and ask it to decrypt it using the symmetric key just unwrapped. The Pilot will decrypt the message, display it on its screen, and send bogus information back to Communicator. So, depending on the preference settings on the Pilot, Communicator may or may not be able to see the decrypted contents of a message[6].

Users that have little trust in the PC that their Pilot is connected to can use this feature to read encrypted email on their Pilot without risking its contents to become known to anyone. This cannot be done with traditional smart cards since they lack a user interface. See Figure 4 to illustrate this point.

[5]Note the difference between "unwrappinng" and "decrypting". An unwrapped key is not returned to the caller (only a reference to it), while a decrypted key is.

[6]If the Pilot agrees to unwrap the key, it will refuse to "decrypt" it, so that even a malicious version of Communicator would not be able to decrypt the message on its own.

3.5 Performance Issues

A Pilot is not as fast as a smart card when it comes to long integer arithmetic. Signing/decrypting a message with a 512 bit key (the key size in exportable versions of Communicator) takes about 5 seconds. Signing/decrypting with a 1024 bit key takes about 25 seconds. (These measurements were made with version 2.01 of pilotSSLeay.)

Creating a 512 bit RSA key pair takes a couple of minutes. Creating a 1024 bit key pair takes 30 minutes. These numbers may vary since a randomized algorithm is used to find the primes.

Communicator often calls into the cryptoki library to reassure that the token is still connected, to exchange information, to search for objects, etc. Through a rather slow serial link, this conversation takes up a lot of time. We built in Communicator-side caches for most of the information that Communicator learns from the Pilot. So what users experience is a flurry of messages going back and forth between PC and Pilot when they start using it during a session. However, after a while the Pilot will only be contacted to actually perform a cryptographic operation, and the time experienced by the users closes in on the numbers given above.

4 Protecting Sensitive Data

How safe is the user's sensitive data if the Pilot gets into the hands of adversaries? In our implementation, sensitive parts of the private key are stored encrypted in nonvolatile RAM. We derive a DES key from the PIN or password that the owner uses for logging on to the Pilot and use it to encrypt the sensitive parts of the private key. Later, they are only decrypted just before they are used, and erased after they are used. When the user logs off, the PIN or password and the derived DES key are erased.

It is very easy for an adversary to read the encrypted key out of non-volatile RAM. He then needs to perform an attack on the DES encryption, or alternatively a dictionary attack on the PIN or password. Since only the actual bits of sensitive data (which are unknown to the attacker) are encrypted, a straightforward known plaintext attack is not possible. Instead, the attacker would have to perform an expensive multiplication to test whether he had found the correct private key. We therefore assume that the private key is reasonably safe, with the weakest link in the chain probably being the PIN or password, which could be too short or part of a dictionary.

For a device like a Pilot, a different threat is more imminent. Our PKCS#11 application usually shares the Pilot with many other applications. Since there is no memory protection, other applications might be able to read the decrypted private key if they manage to interrupt the PKCS#11 application just at the right time. So, for better security users should be very careful about what kind of applications they install on their Pilot.

To alleviate this situation, the PalmPilot would need a comprehensive security architecture. First, the operating system should enforce memory protection and add access control to its resources such as the databases in non-volatile RAM. Second, downloading software onto the PalmPilot should be restricted; we could for example imagine a password needed to download new software onto the Pilot. Third, downloaded software should be put in sandboxes that are enforced by the operating system. There should be a way to relax sandboxes, perhaps based on digital signatures, in order to give certain applications access to more functionality. With a system like this, a user could for example download a game from an unknown source and place it into a tight sandbox from where it cannot possibly learn things about the other applications running.

Since we started this work in 1997, a number of vendors and research projects have tried to address the problem of operating system security in hand-held computers. Recent versions of Microsoft's Windows CE operating system include a feature that allows only applications signed by designated principals to be considered "trusted". Untrusted applications are barred from calling a set of system calls considered sensitive. Sun and 3COM have recently announced that Sun's Java Platform 2, Micro Edition, will be available for the 3COM Palm series of devices. Also, since smart cards themselves have become more powerful, the operating systems on smart cards now tend take into account the scenario of multiple, mutually suspicious applications running on the same card. The techniques used on modern smart cards (secure file systems, etc.) could be applied to hand-held computers.

Once this security architecture is in place, we can turn to physical security: Now that it is impossible for an adversary to inject software into the Pilot, we need to make sure that the data cannot be read out in some other way. To prevent this, the Pilot could be equipped with tamper-resistant hardware, which would make it difficult to obtain the data in question by physical means. Equipping a Pilot with tamper-resistant hardware is not too hard since it is bigger than a smart card and there is more space to accommodate tamper-aware protection mechanisms. But as long as there is an easy way to install arbitrary applications on the Pilot, the additional cost of a tamper-resistant hand-held computer would not be justified.

5 Future Directions

So far, we have described a particular system that implements smart card functionality on a PalmPilot. We have also seen that a hand-held computer has potential for better security, since it provides a direct user interface. The fundamental reason for the desirable security features is that the Pilot is more *trusted* than the PC. Just like a smart card, we always carry it around with us and are fairly certain that no-one tampered with it. We have a good overview of the software running on the Pilot and usually know where it came from. Contrast this with a PC, which may easily run hundreds of different processes at a time. These different processes may open up vulnerabilities to attack. Moreover, if we use our Pilot with an arbitrary PC, we do not know whether that PC has not been specifically prepared to undertake malicious acts.

In this section we are going to explore the possibilities of the following scenario: A trusted, small, moderately powerful computer working together with a more powerful, big, but untrusted, PC to achieve some functionality. The trusted small device is needed to assure a certain amount of security, and the big PC is needed to give a convenient and powerful interface for security-insensitive operations. For example, the PC could display media-rich documents that are not sensitive. This is really just a generalization of the case discussed so far in this paper.

As a first example, let us get back to email. Our implementation presented in Section 3 has at least two shortcomings:

1. The user does not see, on the Pilot, the message that is to be digitally signed[7]. This means that a malicious version of Communicator could forge email.
2. Only for encrypted messages can the user decide whether or not they should be displayed on the PC.

A better approach would be if the email application was controlled from the Pilot. The Pilot establishes an encrypted connection – through the PC it is connected to – to the user's mail host. Then the user decides where a certain message should be displayed.

Another possible application is electronic commerce. Here, also, smart cards are often used. However, because of the lack of an interface, we do not really know what our smart card is doing and how much money it is spending. With a Pilot, the picture looks different: We again consider a scenario where three players are involved. First, there is a server offering goods to purchase. In addition, there is a trusted Pilot (or similar device) which is connected to an untrusted PC. The Pilot carries our electronic cash. On the PC, we launch the client application that connects to the server and displays the offered goods. The Pilot also authenticates itself to the server and establishes an encrypted channel that the PC cannot decrypt.

The client application can be used to view the items, get price information, initiate a purchase, etc. Before the Pilot spends any money, it displays relevant information such as the price and a description of the product on its screen. Only if and when the user confirms this *on the Pilot* can the transaction proceed. We note that even if the server and PC collaborate, the Pilot, which acts as an electronic "wallet", will not dispense more money than the user acknowledges.

Generalizing from these examples, we envision a new programming paradigm we call *Splitting Trust*. Under this paradigm, applications are split to run on different devices. Part of the application runs on a small, trusted device, and part of the application runs on a bigger, more powerful, but untrusted, device. We believe that this splitting enables users to get both security and computing power. They get security because a crucial part of their application runs on a trusted device. They get computing power because the more powerful device can be used to run non-crucial parts of the application. Part of our future work will be to provide middleware to enable easy splitting of applications in this fashion.

[7]Communicator never passes the message content into the cryptoki library. Rather, it calculates the hash itself and then just lets the cryptoki library sign the hash of the message.

6 Conclusions

In this paper we have argued that small hand-held computers can be used instead of smart cards. Moreover, they provide a direct interface to the user. We implemented a PKCS#11 library for Netscape Communicator and corresponding smart card functionality for a 3COM PalmPilot.

In our implementation, the PalmPilot provides a trusted authentication path and gives the user a choice where an encrypted email message should be displayed: in Communicator on the PC or on the PalmPilot itself. This increases the user's privacy above the level provided by traditional smart cards.

We also propose to generalize from our experience and to introduce a new programming paradigm. Under this new paradigm, applications are split into two parts: One part runs on a trusted, but small and only moderately powerful, device; the other part runs on a bigger, more powerful, but possibly untrusted, device like a PC in a public place. Splitting applications will give us both certain security assurances on one hand, and convenience and speed on the other hand. We plan to provide middleware to assist the process of splitting applications.

7 Related Work

In [Yee, 1994] Yee introduces the notion of a *secure coprocessor*. A secure coprocessor is a tamper-resistant module that is part of an otherwise not necessarily trusted PC. Certain goals (notably copy protection) can be achieved by *splitting* applications into a *critical* and *uncritical* part; the critical part runs on the secure coprocessor. While this idea is very similar to ours, the context is different: In the world of secure coprocessors the user is not necessarily trusted (the coprocessor secures information from, among others, the user using it). On the other hand, the user always trusts the secure coprocessor, even if it is part of an otherwise unknown PC. In our world, the user/owner of the PalmPilot is trusted by definition (the whole point of our design is to protect the user). The PC, or any of its parts (even if it looks like a secure coprocessor) is never trusted.

Gobioff et al. notice in [Gobioff et al., 1996] that smart cards lack certain security properties due to their lack of user I/O. They propose that smart cards be equipped with "additional I/O channels" such as LEDs or buttons to alleviate these shortcomings. Our design meets their vision, but we come from the opposite direction: We take a hand-held computer that already has user I/O and implement smart card functionality on it.

Boneh et al. implemented a electronic cash wallet on a PalmPilot, which is quite similar to what we describe in Section 5 [Boneh and Daswani, 1999].

Cryptographers have dealt with the "splitting trust" scenario for some time now, even though the work is often not presented from that perspective. For example, Blaze et. al [Blaze, 1996, Blaze et al., 1998] want to use a powerful PC in concunction with a smart card for symmetric key encryption because the PC provides higher encryption bandwidth. However, the PC is not trusted to learn the secret key. "Function hiding" work (e.g. [Sander and Tschudin, 1998]) is usually presented from a persective of copyright protection, but essentially it is also an instance of splitting trust. Boneh et al. use results from multi-party threshold cryptography to speed up RSA key pair generation on the PalmPilot with the help of an untrusted server, which participates in the key pair generation, yet will not learn the private key (see [Modadugu et al., 1999]).

8 Acknowledgments

We would like to thank Ian Goldberg for porting SSLeay to the PalmPilot, enhancing Copilot, and providing valuable Pilot programming tips. We also extend our thanks to Bob Relyea for help with some PKCS#11 details.

References

[Blaze, 1996] Blaze, M. (1996). High-bandwidth encryption with low-bandwidth smart cards. In *Proceedings of the Fast Software Encryption Workshop*, volume 1039 of *Lecture Notes in Computer Science*, pages 33 – 40. Springer Verlag.

[Blaze et al., 1998] Blaze, M., Feigenbaum, J., and Naor, M. (1998). A formal treatment of remotely keyed encryption. In *Proceedings of Eurocrypt '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 251 – 265. Springer Verlag.

[Boneh and Daswani, 1999] Boneh, D. and Daswani, N. (1999). Experimenting with electronic commerce on the PalmPilot. In *Proceedings Eurocrypt '99*, volume 1648 of *Lecture Notes in Computer Science*, pages 1 – 16. Springer Verlag.

[DES, 1993] DES (1993). *Data Encryption Standard.* National Institute of Standards and Technology, U.S. Department of Commerce. NIST FIPS PUB 46-2.

[DSA, 1994] DSA (1994). *Digital Signature Standard.* National Institute of Standards and Technology, U.S. Department of Commerce. NIST FIPS PUB 186.

[Dusse et al., 1998a] Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., and Repka, L. (1998a). *S/MIME Version 2 Message Specification*. IETF - Network Working Group, The Internet Society. RFC 2311.

[Dusse et al., 1998b] Dusse, S., Hoffman, P., Ramsdell, B., and Weinstein, J. (1998b). *S/MIME Version 2 Certificate Handling*. IETF - Network Working Group, The Internet Society, RFC 2312 edition.

[Freier et al., 1996] Freier, A. O., Karlton, P., and Kocher, P. C. (1996). *The SSL Protocol Version 3.0.* IETF - Transport Layer Security Working Group, The Internet Society. Internet Draft (work in progress).

[Gobioff et al., 1996] Gobioff, H., Smith, S., Tygar, J. D., and Yee, B. (1996). Smart cards in hostile environments. In *Proceedings of The Second USENIX Workshop on Electronic Commerce*, Oakland, CA.

[Gutmann, 1997] Gutmann, P. (1997). How to recover private keys for microsoft internet explorer, internet information server, outlook express, and many others - or - where do your encryption keys want to go today? `http://www.cs.auckland.ac.nz/~pgut001/pubs/breakms.txt`.

[Modadugu et al., 1999] Modadugu, N., Boneh, D., and Kim, M. (1999). `http://theory.stanford.edu/~dabo/abstracts/RSAgenkey.html`.

[PKCS#11, 1997] PKCS#11 (1997). *PKCS#11: Cryptographic Token Interface Standard, Version 2.0.* RSA Laboratories.

[PKCS#11, 1998] PKCS#11 (1998). *Implementing PKCS#11 for the Netscape Security Library.* Netscape Communications Corporation, Mountain View, California. `http://developer.netscape.com:80/docs/manuals/security/pkcs/pkcs.htm`.

[Rivest, 1998] Rivest, R. (1998). *A Description of the RC2(R) Encryption Algorithm.* IETF - Network Working Group, The Internet Society. RFC 2268.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. M. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

[Sander and Tschudin, 1998] Sander, T. and Tschudin, C. (1998). On software protection via function hiding. In *In Proceedings of the Second Workshop on Information Hiding*, Lecture Notes in Computer Science. Springer Verlag.

[Schneier, 1996] Schneier, B. (1996). *Applied Cryptography*, chapter 15.2 Triple Encryption, pages 358–363. John Wiley.

[Yee, 1994] Yee, B. S. (1994). *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University.

[Young et al., 1998] Young, E. et al. (1998). SSLeay and SSLapps. `http://psych.psy.uq.oz.au/~ftp/Crypto/`.

Offline Delegation[*]

Arne Helme[†] Tage Stabell-Kulø
Department of Computer Science
University of Tromsø, Norway
{arne,tage}@acm.org

Abstract

This article describes mechanisms for offline delegation of access rights to files maintained by a distributed "File Repository". The mechanisms are designed for a target environment where personal machines are used at times when critical services, such as authentication and authorization services, are not accessible. We demonstrate how valid delegation credentials can be transferred verbally without the use of shared secrets.

Our main result shows that delegation of access rights can be accomplished in a system that uses public-key encryption for secrecy and integrity, without forcing the user to rely on a trusted third party, and without requiring connection to the infrastructure.

The implementation runs on a contemporary Personal Digital Assistant (PDA); the performance is satisfactory.

1 Introduction

When personal machines are incorporated into distributed systems, privacy becomes important. In our view, the mere existence of personal machines implies a new direction in computer security research. Rather than solely protecting centralized resources from unauthorized access, protecting the interests of the individual becomes the focus. For example, users will want to decide for themselves when and to whom access to their resources should be granted. In contrast, consider a system with centralized control, such as Kerberos [Steiner et al., 1988]. In Kerberos, the trust relation between the system and its users is asymmetric and the organization maintaining the system dictates when and to whom access to resources should be granted. It is impossible for a user to generate credentials, either inside or outside Kerberos, that will be valid outside of its realms. Consequently, users cannot delegate access to their own resources at will. This lack of control over personal resources is an architectural concern, and not in any way related to the cryptographic technology that is applied in the system (symmetric or asymmetric cryptography).

Armed with a Personal Digital Assistant (PDA), users will challenge centralized models of authentication and access control by demanding to be in authority of their own resources. In essence, a PDA can provide the user with a Trusted Computing Base (TCB) [Department of Defense, 1985]. The TCB gives leverage in situations where the user accesses resources remotely or wants to delegate access rights to other users.

A system that relies on PDAs for a part of its security creates a new set of engineering challenges. This article examines a problem that is rather specific to mobile computing: how to cope with circumstances where there is low or no connectivity between a user's machine and the system components critical to some operations. Our focus is on how authority can be delegated from one user to another without the requirement of communication with any kind of server.

PDAs are often not connected to a computer network. Even so, delegation of authority should be possible. Delegation may have to take place indirectly and possibly over unconventional paths — such as part of a telephone conversation. Consider the following scenario: Alice and Bob are having a conversation on the phone, and Alice wants to grant Bob access to a file of hers. Since she has her TCB at hand, she should be able to generate sufficient credentials to enable Bob to access the file. This problem is denoted "offline delegation of access rights" (initially described in [Helme and Stabell-Kulø, 1996]). We describe how the problem comes about and manifests itself, what the implications are, and presents a so-

[*]Funded by the GDD project of the Research Council of Norway (project number 112577/431)
[†]Funded by the GDD-II project of the Research Council of Norway (project number 1119400/431)

lution to it. Some relevant details of an implementation are also presented.

The remainder of this article is structured as follows. Section 2 starts out by arguing why offline delegation is desirable, and presents the environment in which a solution has been implemented. Section 3 discusses the security constraints while Section 4 discusses the protocols that are used. Section 5, describes some relevant implementation details. Then, Section 6 discusses aspects of the TCB, the assumptions made about the PDA and the (secure) channels that are present in the system. Finally, the conclusions are drawn in Section 7.

2 Overview

The setting is one with a file repository (called FR) that manages replication and concurrency control of files [Stabell-Kulø and Fallmyr, 1998]. FR has been designed to support users with a variety of equipment, ranging from PDAs to workstations. FR is the research vehicle used to investigate the thesis that users should be involved at the places in a system where decisions (either implicit or explicit) are made. In particular, in a system with PDAs disconnection will be common and consistency problems occur frequently; FR has been designed to allow users to deal with them in any way they choose.

Along the same line of thought: Why should users be forced to contact FR simply to generate a delegation certificate? Stated differently: Is delegation only possible when there is connectivity? Systems that are constructed so that principals must be on line for delegation to take place, effectively exclude PDAs. How to delegate authority without connectivity is the theme central to offline delegation.

In FR, users are represented as principals by their public keys. Consequently, all communication channels can be authenticated for integrity and encrypted for privacy. An integrated part of FR´s design is that authority over files can be delegated freely. More specifically, a delegation certificate names a file and a user, together with an access right (read, write or both), and the time of creation (of the certificate) and when it will expire. The syntax and semantics of delegation certificates are discussed in detail below.

Making offline delegation possible requires that two problems are solved. First, one must be able to generate a valid certificate by means of a PDA without having access to FR. Second, it should be possible to convey the certificate verbally; without a computer network messages must be sent by means of the communication channel that is available; that is, by means of human speach. This constraint rules out every binary representation of certificates in so far that it is unlikely that anyone will be able or willing to read hundreds of digits over the phone. Similar arguments also rule out the use of digital signature schemes relying on long signatures (RSA [Rivest et al., 1978], for example, with 1024–2048 signature bits).

The following sections describe both the cryptographic techniques and the tools required in order to construct a practical solution to the problems described above.

3 Security considerations

Design of an offline delegation mechanism must carefully trade off convenience against availability without compromising the overall system security. This section explores the design space, and in particular the requirements for the delegation certificates. A solution must fulfill the following design criteria:

1. A delegation (i.e., a certificate) should not enable any principal to impersonate the delegator or delegatee.
2. The credentials must form valid and meaningful access rights. In particular, all objects (principals, machines and files) must be unambiguously named.
3. The authority granted by a signed statement should not be transferable, and a certificate containing the signature in question should be valid for usage only once.

To ease the task of verbally transferring access rights—while retaining security—the following strategy is used. Generally, of all the information in a typical certificate only the digital signature constitutes binary data. Because delegation certificates contain a wealth of information that can easily be read out, such as dates, file names, Internet addresses, domain names, etc, the amount of binary information that needs to be exchanged is limited. The key issue is then becomes to simplify and ease the exchange of the signature bits.

In order to delegate access rights based on digital signatures, the number of bits that needs to be exchanged is critical to the security of the system. Shared-key systems have shorter keys (64–128 bits) than public-key systems, but can not be used in this system, because they undermine the entire security regime in FR [Helme and Stabell-Kulø, 1997]; we elaborate on this in Section 6. Decreasing the length of the signature then implies using a crypto system with a denser key space. We have opted for the latter, and chosen a crypto system based on elliptic curves in finite fields. For a general description of elliptic-curve cryptography, see [IEEE, 1997, Menezes et al., 1996].

In addition to the requirements described above, to ensure that "once only" semantics can be enforced, each certificate must be unique. We know from experience that it takes (much) more than one second to generate a certificate, so rather than adding a serial number to them, we add the time of creation. With a resolution of the clock of (less than) one second, all certificates (from one principal) will be unique. To some extent, the time of creation can be viewed as a non-contiguous serial number. Including in each certificate its time of expiration, on the other hand, ensures an implicit revocation of old (unused) certificates.

4 Protocol description

This section describes the protocols used to delegate access rights from one user to another. In order to obtain a copy of a file a user must interact with FR by means of a protocol. The protocol is named File Repository Transport Protocol (FRTP) [Stabell-Kulø, 1995]. It resembles the Simple Mail Transport Protocol (SMTP) and Network News Transport Protocol (NNTP) protocols in that all commands and responses are encoded in ASCII with short numerical status-codes being returned for each command.

FR requires that users are authenticated and it provides users with the means to establish the authenticity of the server. At connection setup time, a secure channel is established between the user and FR. Certificates can be presented to FR without having to be signed by the delegatee when sent on an authenticated channel. The certificate can also be signed and subsequently sent unencrypted. In the latter case, the file will be returned on the same connection (but not on a secure channel). The latter approach is naturally equivalent with the former except that secrecy is not achieved. FR supports both kinds of interactions mentioned above.

4.1 Certificate creation

Before a protocol run can take place the delegator (file owner) creates a delegation certificate. This delegation certificate contains a digital signature that vouches for his delegation of access rights for a particular file to the other user. The certificate grants the delegatee access to the file. The certificate components that can easily be conveyed verbally are the following:

- the names of the file, the delegator and the delegatee,
- time of creation and expiration
- specification of delegated access rights. In the current design, access rights can be either *read* or *write*. A certificate will, thus, enable the delegatee to access the file according to the delegated access rights specified in the delegation certificate.

The verbal delegation ends with the exchange of the bits resembling the signature. Together with the other information exchanged, the signature enables the delegatee to (re)construct a machine readable representation of the certificate. These certificates, obviously, must be readable not only by computers but also by humans. To facilitate this, certificates are encoded in a syntax similar to SDSI [Rivest and Lampson, 1996].

4.2 Access-request protocol

The *access-request* protocol performs a forward authentication of access rights from the delegator to the server via another user (the delegatee). In short, to access a file the delegatee signs the delegation certificate with his own private key and transmits the result to the file server (FR). The following messages are exchanged:

Message 1 $A \rightarrow B: \{A,B,F,AC,T,S\}_{K_A^{-1}} \quad (=X)$

Message 2 $B \rightarrow S: \{B,S,X,WD\}_{K_B^{-1}} \quad (=Y)$

Message 3 $S \rightarrow B: \{S,B,H(X),H(Y),RD,WD\}_{K_S^{-1}}$

In the protocol description, A and B represent the users Alice and Bob, S is the FR (server), F is the name of the file in question, AC are the delegated access rights, and T are two time stamps, one making the certificate unique the other ensuring that the certificate expire. K_A^{-1} is the private key belonging to A. Message 1 (or X, in short) is the delegation certificate. $H(X)$ is the message

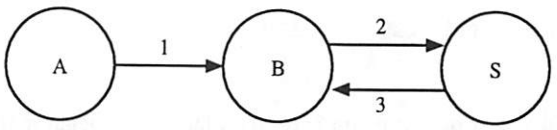


Figure 1: Messages in the protocol

digest of X, and is essentially B's receipt from S. The field *RD* is data that has been read while *WD* is data to be written. If the access-right is *read*, *WD* must be *nil*. The protocol does not distinguish between conventional (networked) or verbal transfer of the first message — in both cases the same information is presented to the server in Message 2.

4.3 Analysis

The protocol consists of three messages, as illustrated in Figure 1. The messages are explicit, and contain sufficient information that their meaning can be established without context. That is, the meaning of messages does not depend on a (set of) previous message(s).

A BAN logical analysis of the protocol begins with the observation that Message 1 (denoted X in the protocol description) is received by B over a "real time" channel, implying that $B \mid\equiv \sharp(X)$ (see [Abadi et al., 1991]). The assumption $B \mid\equiv \overset{K_A}{\mapsto} A$ makes it possible for B to believe that $A \mid\sim X$; a similar argument also holds for S. If B believes that S is honest [Syverson, 1992], we obtain $B \mid\equiv S \mid\equiv data$ because $B \mid\equiv \sharp(X)$ and $B \mid\equiv S \mid\sim (X, data)$ (derived from Message 3). In other words, B believes that he has received the correct file from S.

At a higher level [Lampson et al., 1992], Message 1 contains two statements (we only deal with reading, writing is similar):

$$A \textbf{ says } B|A \Rightarrow B \textbf{ for } A \tag{1}$$

and

$$A \textbf{ says } B \textbf{ for } A \textit{ may read file } F. \tag{2}$$

Statement (1) says says that the combined principal $B|A$ may speak for B **for** A, while (2) says that B **for** A may read from file F. When B includes Message 1 in Message 2, it is interpreted as

$$B|A \textbf{ says } \textit{read } F. \tag{3}$$

Statement (3) enables S to deduce that B (by quoting A) indeed speaks for B **for** A. Consequently, if A owns the file F, then (3) should be honored.

5 Implementation details

This section describes the implementation of the offline delegation mechanism in greater detail. The implementation of the File Repository currently runs on Unix and Windows NT. As PDA we have used the Palm-III from 3Com with PalmOS as the operating system. The offline delegation mechanism is hosted both on the PDA and on Unix workstations.

In addition to FR itself, the implementation consists of a library with cryptographic functions, a library to parse and generate SDSI objects and a graphical user interface to create/send and receive/verify certificates on the PDA. First we will describe the user interface of the offline-delegation application running on the PDA. It is by means of this application that the users can *create*, *send* and *receive* certificates. Then we will focus on the two libraries that are part of our implementation. The first library contains cryptographic functions to create short digital signatures. The second library converts certificates between internal and external representations.

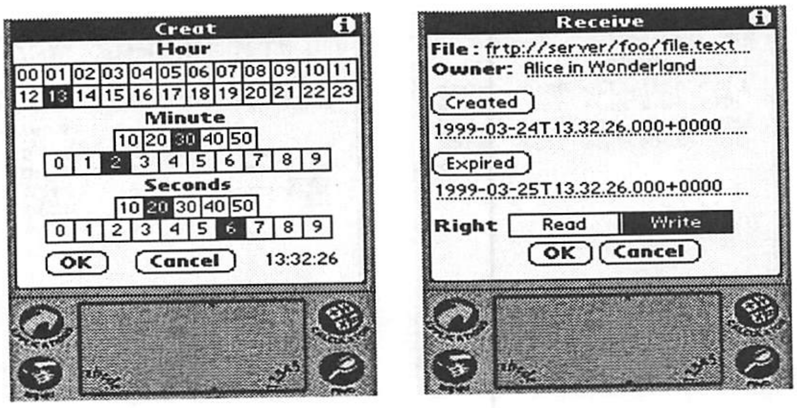
5.1 User interface

The bandwidth of the channel defined by human conversation is relatively low. The core of an offline delegation system is the dual ability to generate a valid certificate on a PDA and making it "readable" in a form which is simple to both send and receive on such low-bandwidth communication channels. To facilitate offline delegation, we have designed and implemented a supporting application on the PDA. The appearance is such that it enables two users to exchange sufficient information to send a certificate on one end, and receive it on the other.

In general, the process of building a certificate consists of entering the required information into the application, and then let the application generate it. The following information is required:

Filename: It is the users' responsibility to ensure that file names are unique within the scope of a server; the name of the file is prefixed by the name of a server. In the implementation, applications synchronize with the file server and has a cache of filenames available. If access is delegated to a file which name is not available, the name must be entered manually.

Delegatee: In all systems where users are represented by their public keys, names must be unique within



File : frtp://server/foo/file.text
Owner: Alice in Wonderland
1999-03-24T13.32.26.000+0000
1999-03-25T13.32.26.000+0000
Right Read [✓ Write]

(a) Setting the time

(b) Information

Figure 2: Elements from the user interface components to create delegation certificates

the system. More precisely, the name appearing in the certificates must be unique. As with file names, the software keeps a cache of frequently used user names.

Created, Expire: In order to ease the process of entering dates, calendars are used (with the current date as default). Figure 2(a) shows how time is entered into the application; the user taps the boxes with the stylus.

Rights: A certificate can delegate authority [Lampson et al., 1992]. In our setting, such authority can be read or write (or both).

Figure 2(b) shows how general information is entered into the application.

The assumed operational procedure is that Alice enters information into the data fields on her PDA while she talks with Bob, and Bob enters the same information into his PDA. The software has been designed to be used in this way, that is, the order in which information is entered when a certificate is created is matched on both sides. By going through the fields together, they build up the certificate. When Alice is finished entering information, she will sign the data with her private key.

As will be explained below, the signature is 256 bits long. On the sending side, the bits are presented to Alice as 16 4-digit hexadecimal numbers; see Figure 3(a). Notice that the checksum in the right hand column is a simple error detection scheme. She can now read the signature bits out to Bob, one group at a time.

As Bob listens to Alice, he needs means to enter the signature bits as fast as Alice reads. To facilitate this, a dedicated form is presented on the PDA. By tapping on the screen of the PDA he is able to enter data (*receive* as it were). The design is such that users can receive bits fast enough for the system to be usable. See Figure 3(b) for a signature that has been partly received.

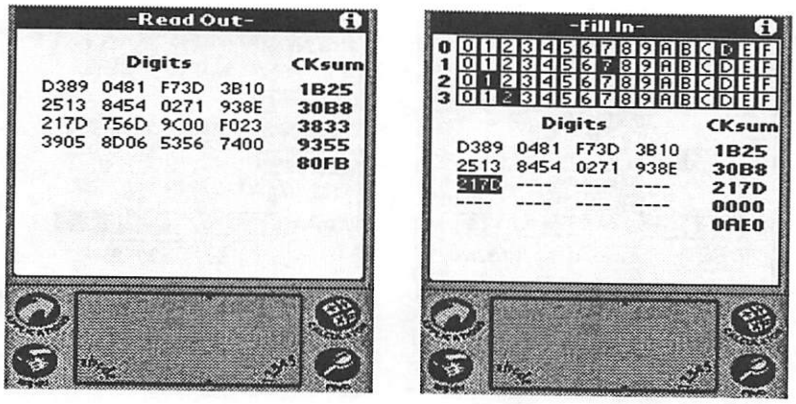
The checksum is calculated as data is entered. Although only a proper verification of the signature can determine whether it was properly transferred, the checksum is used to give Alice and Bob some confidence in its integrity. In this implementation the checksum is an exclusive-or function computed over the four 16-bit numbers. As a result, two bits in any number of row of bits will go undetected. If experience shows that a stronger integrity check is required, it can easily be incorporated into the application.

In the Unix client, the method used in the S/KEY one-time password system [Haller, 1994] has been adopted and the signature bits are conveyed in the form of English words.

5.2 Crypto library

The cryptographic library provides functions to create and verify digital signatures. Currently the library is available on both Unix and PalmOS platforms. The library provides the Nyberg-Rueppel version of elliptic curve signatures [Nyberg and Rueppel, 1993] and the SHA1 message digest function.

The implementation of elliptic curve cryptography is



```
       Digits              CKsum
D389  0481  F73D  3B10     1B25
2513  8454  0271  938E     30B8
217D  756D  9C00  F023     3833
3905  8D06  5356  7400     9355
                           80FB
```

```
0 0 1 2 3 4 5 6 7 8 9 A B C D E F
1 0 1 2 3 4 5 6 7 8 9 A B C D E F
2 0 1 2 3 4 5 6 7 8 9 A B C D E F
3 0 1 2 3 4 5 6 7 8 9 A B C D E F
       Digits              CKsum
D389  0481  F73D  3B10     1B25
2513  8454  0271  938E     30B8
217D  ----  ----  ----     217D
----  ----  ----  ----     0000
                           0AE0
```

(a) Sending

(b) Receiving

Figure 3: Elements from the user interface to send and receive delegation certificates

based on the algorithms for fast operations in finite fields. [Win et al., 1996] that has been tuned to fit the limited processor and memory resources on a small PDA. The current implementation uses a finite field of order $GF(136)$ with fast operations on elements in a subfield of order $GF(8)$. The order of the fixed point on the curve is a 241 bit prime number. This yields Nyberg-Rueppel scheme digital signatures with a total length of 256 bits for both components of the signatures.

The security of the elliptic-curve signature scheme relies on the difficulty of finding discrete logarithms in finite fields of special form. With the current field size and chosen curve parameters the security of the system is estimated to be at least of similar strength as 1024 bits RSA signatures. Digital signature schemes based on elliptic curves show not only promising results with respect to performance (signing speed, in particular), but also when it comes to strength per bit and memory utilization [IEEE, 1997].

5.3 SDSI library

All keys, certificates and protocol messages used in the implementation of the offline delegation mechanism are specified in a format similar to SDSI [Rivest and Lampson, 1996]. Only the syntax specification of SDSI has been adopted, however, and not its associated public-key infrastructure. We have chosen an external representation that is "human readable". That is, the majority of data in the certificate is represented in ASCII. In our experience it is valuable to be able to "look at" certificates to compare the fields one by one.

SDSIlib is based on the SEXP library designed and implemented by Ronald Rivest. The library has been extended to support a syntax similar to SDSI and contains sufficient functionality to build parsers and generators for new protocols with SDSI encoding of the protocol messages. The SDSIlib API [Helme, 1998] specifies the external representation of data in SDSI objects, and a set of library functions to manipulate such objects. The library contains basic functions to parse and generate basic SDSI objects. In the SDSIlib port for the Palm-III, most of the library's functionality has been retained. The library can be configured to read and write SDSI objects from and to TCP/IP streams, Unix-like file I/O, or PalmOS databases.

As an example, consider the specification of a `Signature` object containing a signed delegation statement in Figure 4. The `Signature` object contains an offline delegation certificate. More specifically, the signed object consists of the information that has been signed (`Object`), and information (`Algorithm`) about the signature algorithm that has been used to create the signature in question. The `Algorithm` field also holds information about the hash algorithm and signature algorithm that have been used to create the digital signature. The format of the `Algorithm` field is *signature-algorithm*`-with-`*hash-algorithm*. The name of the algorithm-dependent hash field is *hash-algorithm*`-Hash`. The name of the algorithm-specific signature field is *signature-algorithm*`-Signature`. The fields `Object-ref` and `Object-perms` identify the file subject to delegation and the delegated access rights.

```
( Signature:
  ( Object:
    ( Delegate-From: "Alice in Wonderland" )
    ( Delegate-To: "Bobs Country Bunker" )
    ( Object-ref:  frtp://server/foo/file.text )
    ( Object-perms: read )
    ( Created: "1999-03-24T13.32.26.000+0000" ) )
    ( Expire: "1999-03-24T13.32.26.000+0000" ) )
  ( Algorithm: ECNR-with-SHA1 )
  ( SHA1-Hash: |c+1xi/6oE4k5Hr8JPR1T4Q==| )
  ( ECNR-Signature:
    ( Signing-Key: |B3ZbHXKyBwQ=| )
    ( Galois-Field: #88# )
    ( R: |Gk/PrhBpnNdXinxC11krrQ==| )
    ( S: |DPI7aahT4A9c5MeG7EF/VQ==| ) ) )
```

Figure 4: Specification of a delegation certificate in a syntax similar to SDSI

6 Discussion

This section discusses some aspects of offline delegation in distributed systems, and in our system in particular. First we argue why offline delegation does not weaken the security of the system, then we discuss issues related to the trusted computing base, revocation of certificates and performance of the prototype implementation.

6.1 Security concerns

We believe that the provision of offline delegation does not weaken the security of FR in any way. This hinges on the fact that the file owner is solely responsible for the security policy he implements. The increased flexibility offered by offline delegation comes with the price of responsible and competent users. However, for each delegation there is only a single file involved, and other files are not involved in any way, so a user can not compromise the security of any other user. The security problem intrinsic to stolen PDAs is treated in the section on the TCB.

A file can be handed out erroneously if it is given the same name as an old file, and access to the old file has been delegated. A new file with the same name as an old file can be regarded as if the contents was altered on the old file. FR can thus not distinguish between rightful access to new data in the old file and incorrect access to a new file with the same name as an old file. By ensuring that all certificates expire properly (within a reasonable time frame) names of files can be reused after that time. We do not consider this a security problem.

The security regime of the system is built on public key cryptography. It is a goal that FR should be able to produce a certificate for each and every transaction that takes place. In such a system, a scheme built on shared keys is very hard to conceive. Any secret shared with FR can be used by FR to convince itself about the origin of a message, but such "proof" has no value to others.

6.2 Trusted Computing Base

The single most intriguing issue with the concept of PDAs is the possibility that every user can have a Trusted Computing Base (TCB) under his control which does not include resources controlled by others. That is, if the system is designed in such way that the user's PDA constitutes his TCB, then keys and credentials can safely be stored in it. More specifically, a trusted PDA can act on its owners' behalf when the owner delegates access rights. In particular, when it comes to the case of binding a public key to a human, the TCB consists only of the user's PDA. This, of course, stems from the use of public-key technology rather than offline delegation per se.

Recall the scenario with Alice delegating authority to Bob while speaking on the phone. In this scenario, FR is not part of the TCB because it is "only" used to store files. The result is that FR is unable to impersonate the user, neither when interacting with other instances of FR, nor when interacting with other users.

6.3 Revocation

Disconnected operation is common in our system and revocation of access rights is consequently a concern. Effective revocation of access rights in distributed systems is generally considered a hard problem to solve [Lampson et al., 1992], and lack of connectivity makes the problem even more difficult. This places limits on when revocation can be performed. In order to revoke a certificate there are essentially two approaches: either to

1. limit the time frame in which certificate is valid,
2. let the certificate be valid only once.

Both approaches have their merits and disadvantages [Helme, 1997]. FR provides offline delegation and it supports both mechanisms. To ensure that certificates are used only once, once they are used they are stored by the server until they expire. This policy facilitates that certificates have "once-only" semantics (see [Helme, 1997]). Users can not override this policy, but if there is a need to grant another user access on a more permanent basis, Access Control Lists (ACL) can be implemented. A discussion of ACLs in FR is beyond the scope of this article.

Timestamps are used as an additional source of information for revocation purposes. Since individual users specify access policies, the correctness of the time stamp encoded into each delegation certificate depends entirely on this user's ability to determine what the current time is. However, timestamps are only used to recognize and refuse old certificates. The use of time to discard once-only delegation certificates is not entirely without risks (for a discussion, see, for example, [Gong, 1992]).

6.4 Performance

On the platform for which we have made our implementation, entering information is time consuming. The graphical user interface is pen-based, and writing is rather slow. Some fairly long strings (such as file names with an associated path) have to be entered to uniquely identify a file, or selected from a list of frequently used strings.

On the PDA, to create a digital signature on a certificate takes about 5 seconds while verifying a signature takes about 10–15 seconds. While the process of signing certificates is mandatory, verifying the signature on the PDA is not. The receiver may choose to pass the certificate on to FR without verifying that it is correct. FR runs on Unix, and verification of a signature in this environment takes fractions of a second and does not pose a performance problem in the intended target environment.

6.5 Future work

The File Repository is the research vehicle for several other projects on data consistency and distributed applications, and the offline delegation mechanism will be used by these other projects whenever feasible.

One related project uses the File Repository to hold information about appointments. The offline delegation mechanism enables users to delegate access rights to other users in order to let them either read diary information or to (selectively) update it. The Global Distributed Diary (GDD) project explores these issues further.

7 Summary and conclusions

We have described how offline delegation is a natural extension to the services already offered by FR. The argument is focused on the concept of "user in the decision loop", and offline delegation is a consequence of this design philosophy. Offline delegation is used to delegate access rights from one user to another, in a setting where communication with FR is impossible at the time (or undesirable for some reason). To ease the exchange of delegation certificates, a method has been developed that enables two users to convey a certificate verbally. Cryptographic techniques based on elliptical curve encryption are used to facilitate short signatures so that as little data as possible has to be conveyed while maintaining a high level of security. The increased complexity in the implementation is outweighed by the advantage of short signatures.

The PDA of choice is the Palm-III, manufactured by 3Com. It was shown that performance is satisfactory even on this class of hardware. Furthermore, by utilizing the graphical user interface on the PDA, it is possible to transfer certificates (using speech) as part of a conversation between humans. Security is preserved and the performance is satisfactory.

Acknowledgements

The authors want to thank Feico Dillema, Jaap-Henk Hoepman, Åge Kvalnes, Sape Mullender, Per Harald Myrvang, and the anonymous referees for valuable comments. The first author also wants to thank George Barwood and Paulo Barreto for providing source code and useful information on how to implement elliptic curve cryptography.

References

[Abadi et al., 1991] Abadi, M., Burrows, M., Kaufman, C., and Lampson, B. (1991). Authentication and Delegation With Smart-cards. *Theoretical Aspects of Computer Software*, pages 326–345. Springer-Verlag.

[Department of Defense, 1985] Department of Defense (1985). DoD 5200.28-STD: Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC).

[Gong, 1992] Gong, L. (1992). A Security Risk of Depending on Synchronized Clocks. *ACM Operating Systems Review*, 26(1).

[Haller, 1994] Haller, N. M. (1994). The S/KEY One-time Password System. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, CA.

[Helme, 1997] Helme, A. (1997). *A System for Secure User-controlled Eletronic Transactions*. PhD thesis, Informatica, University of Twente, Enschede, the Netherlands.

[Helme, 1998] Helme, A. (1998). SDSIlib API: Tutorial and Programmer's Reference Manual. In preparation.

[Helme and Stabell-Kulø, 1996] Helme, A. and Stabell-Kulø, T. (1996). Off-Line delegation in a File Repository. In *1996 DIMACS WorkShop on Trust Management in Networks*, Rutgers University. Work presented at workshop.

[Helme and Stabell-Kulø, 1997] Helme, A. and Stabell-Kulø, T. (1997). Security Functions for a File Repository. *ACM Operating Systems Review*, 31(2):3–8.

[IEEE, 1997] IEEE (1997). Standards Specifications for Public Key Cryptography (P1363, Draft Version 7). IEEE Standards Draft.

[Lampson et al., 1992] Lampson, B., Abadi, M., Burrows, M., and Wobber, E. (1992). Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310.

[Menezes et al., 1996] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Series on Discrete Mathematics and Its Applications. ACM Press.

[Nyberg and Rueppel, 1993] Nyberg, K. and Rueppel, R. (1993). A New Signature Scheme Based on the DSA Giving Message Recovery. In *First ACM Conference on Computer and Communications Security*, pages 58–61. ACM Press.

[Rivest et al., 1978] Rivest, R., Shamir, A., and Adleman, L. (1978). A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120–126.

[Rivest and Lampson, 1996] Rivest, R. L. and Lampson, B. (1996). SDSI—A Simple Distributed Security Infrastructure. `http://theory.lcs.-mit.edu/~rivest/sdsi10.ps`. (Version 1.0).

[Stabell-Kulø, 1995] Stabell-Kulø, T. (1995). File Repository Transfer Protocol (FRTP). Technical Report CS-TR 95-21, Department of Computer Science, University of Tromsø, Norway.

[Stabell-Kulø and Fallmyr, 1998] Stabell-Kulø, T. and Fallmyr, T. (1998). User controlled sharing in a variable connected distributed system. In *Proceedings of the seventh IEEE International Workshop on Enabeling Technologies: Infrastructure for Collaborative Enterprises (WETICE'98)*, pages 250–255, Stanford, California, USA. IEEE Computer Society.

[Steiner et al., 1988] Steiner, J. G., Neumann, B. G., and Schiller, J. I. (1988). Kerberos: An Authentication System for Open Network Systems. In *Proceedings of the Winter 1988 Usenix Conference*, pages 191–201.

[Syverson, 1992] Syverson, P. (1992). Knowledge, Belief and Semantics in the Analysis of Cryptographic Protocols. *Journal of Computer Security*, 1(3):317–334. IOS Press.

[Win et al., 1996] Win, E. D., Bosselaers, A., Vandenberghe, S., Gersem, P. D., and Vandewalle, J. (1996). A Fast Software Implementation for Arithmetic Operations in $GF(2^n)$. In Kim, K. and Matsumoto, T., editors, *Advances in Cryptology – ASIACRYPT'91*, volume 1163 of *Lecture Notes in Computer Science*, pages 65–76. Springer-Verlag.

Vaulted VPN: Compartmented Virtual Private Networks On Trusted Operating Systems

Tse-Huong Choo
Hewlett-Packard Laboratories, Bristol, United Kingdom

Abstract

Virtual Private Networks for IPSec based on an intermediate packet-redirector in network-protocol stacks are becoming increasingly common for many standard operating systems and represent a well-understood method for retro-fitting such systems with IPSec support. This report describes how a different design structured around a Trusted Operating System can offer better security, performance and robustness. We describe in detail an implementation of an IPSec VPN consisting of a series of compartmented, concurrently executing IPSec stacks. The motivations and security-related benefits behind each design decision are discussed. In addition, we show how a configuration of independent IPSec stacks based on this design can be configured to execute in parallel for greater performance on single-threaded kernels, and how its design allows individual component-failures without affecting the system as a whole.

1 Introduction

There is an increasing prevalence of IPSec Virtual Private Networks (VPNs) based around an approach of inserting a plug-in into the network protocol stack of mainstream operating-systems. This plug-in typically redirects any incoming or outgoing network packet in order to perform IPSec-specific processing such as encryption or decryption. This approach is well understood and has been mentioned in many standard texts, most notably the IETF RFCs for IPSec [2][3].

However, this approach is not without its weaknesses. The important, but often conflicting design goals of security, performance and robustness often leads to cases where the development of one goal requires trade-offs in the other. For example, placing IPSec-processing in the kernel or operating system level is desirable performance-wise because of copy-avoidance of packets between kernel and user-mode spaces. However, the kernel or operating system can be compromised if an error occurs in the network protocol stack.

This report attempts to describe another approach that seeks to avoid some of these issues by compartmentalizing the components of a traditional VPN into isolated, independent processes. In particular, we aim to show how a more secure version of a conventional VPN can be built to make use of the security-features present in a trusted operating system and also satisfy the goals of high-performance and robustness.

The techniques used in designing a more secure VPN will typically introduce performance degradations that are otherwise not present on a standard VPN-design. For example, the general principle of reducing a monolithic system to a series of smaller, trusted components involves added overhead in synchronization between these components. Therefore, the issues of security and performance have to be considered as inextricably linked. These two issues, together with robustness, is fundamental to many of the topics presented here.

This report describes the design and operation of the Vaulted VPN, which is a software-based IPSec VPN built on a slightly modified version of the HP-UX 10.24 Virtual Vault Operating System. We begin by describing briefly the security-features of HP-UX 10.24 relevant to the Vaulted VPN and continue by describing the architecture and operation of the Vaulted VPN, followed by a security analysis of each aspect of VPN's operation.

We also show how compartmented VPNs such as the Vaulted VPN can take advantage of parallel processing and how they can continue to function even if some portions fail – not an unimaginable occurrence given the many well-documented attacks using malformed IP packets [19] [20]. This represents an unexpected level of robustness considering the monolithic nature of most network protocol stacks.

Although this report mentions the use of HP-UX 10.24 specifically as a target platform, the principles shown here could be applied to full-blown CMW platforms [9] such as HP-UX 10.26 or Sun Microsystems' Trusted Solaris 2.5.

2 Features of the HP-UX 10.24 Virtual Vault Operating System

The target platform of the Vaulted VPN is the HP-UX 10.24 Virtual Vault Operating System. The HP-UX 10.24 operating system is a UNIX-style Multi-Level Secure (MLS) operating system that implements additional security-features beyond those normally available on standard UNIX systems. It is a CMW-derived operating system [6] [9] and features the Mandatory Access Controls (MAC) and principles of least-privilege found on full CMW-type systems, but differs in that it lacks the Trusted Windowing and Trusted Networking features of a full CMW-type system. This section describes the security-features used by the Vaulted VPN.

2.1 Mandatory Access Control

In addition to the standard UNIX-style security features, HP-UX 10.24 also implements a Mandatory Access Control policy based on the Bell-LaPadula formal model [11]. Mandatory Access Control (MAC) is a set of rules that govern how data may be accessed on the trusted system. The policy is called mandatory because users cannot choose which data will be regulated under this policy – the trusted system enforces MAC consistently. In this model, all resources or objects in the system are given sensitivity labels, which consists of a classification and a number of compartments. Access to an object is controlled by its sensitivity label. To have read-access, a process must have a sensitivity label which 'dominates' that of the object being read. For write-access, the labels must match exactly.

2.2 Discretionary Access Control

HP-UX 10.24 has discretionary access controls (DAC) which are similar to those found on standard UNIX systems, specifically the read-write-execute mode-bits based on user and group identities. It is most effective when used in conjunction with Mandatory Access Control, for example, when used to restrict access to everyone but a special pseudo-user used by specific trusted processes.

2.3 Least Privilege

On a traditional UNIX system, the ability to override security controls rests entirely within the root-account. On trusted systems, this all-powerful ability has been broken down into a large number of individual privileges, each of which confers its holder the ability to perform a specific action. Every action that could affect security is restricted with a named privilege, and only processes with the appropriate privileges are allowed to perform these actions. Following the principle of least privilege, this finer level of granularity allows specific programs to be assigned with the minimum-required set of privileges in order for them to perform their assigned tasks.

3 IPSec Protocol

A significant fraction of commercial VPNs are based on IPSec technology which defines the encoding of the encrypted IP packets. Normal unencrypted IP packets are made up of an IP-header followed by a variable length payload in clear-text which may itself include headers for higher-level protocols, e.g. TCP or UDP.

IPSec packets appear to be normal IP packets to switching and routing hardware, but feature internal payloads which may be encrypted, authenticated or both. There are two formats used: the AH format [5] which allows data to be authenticated using a keyed hash-function, and the ESP format [4] in which data is encrypted and optionally authenticated, depending on which encryption algorithm is selected.

In addition, IPSec allows for two main modes of operation, namely transport and tunnel modes. Tunnel-mode allows entire IP packets of arbitrary protocols to be encapsulated within another IP header to allow for authentication of the original IP packet's header, and is typically used in creating an encrypted tunnel between gateways.

The IPSec protocol may be imagined to be operate at a level below or at the IP-layer of network protocol stacks. The secret keys used to encrypt the payloads are derived earlier using some higher key-agreement protocol (such as IKE [1]) or simply manually keyed.

4 Vaulted VPN Architecture

This section gives an overview of the architecture of the Vaulted VPN by describing the purpose and function of each of its components. An explanation is given on how packets are routed in and out of the kernel, where IPSec processing takes place and how Internet Key Exchange (IKE) [1] negotiations are triggered to form IPSec Security-Associations.

4.1 Design of Conventional Bump-in-the-stack VPNs

First, we consider the design of a conventional bump-in-the-stack VPN. Because the kernel on most operating systems is usually not re-entrant and not directly accessible by user-processes, it is usually designed in a monolithic fashion. The network protocol stack in such kernels is similar – packets flowing up or down the stack from various processes (some sensitive, some not) all go through the same monolithic stack. Therefore, a bump-in-the-stack VPN based on such systems would naturally share the problems of re-entrancy and lack of segmentation.

However, the nature of the HP-UX 10.24 operating system allows a finer level of granularity when handling network packets. Since HP-UX 10.24 applies sensitivity-labels to most objects and resources in the system (including files, processes and network packets), it becomes possible to segment the data present in such a system based on its sensitivity label. Since each process is placed at a given sensitivity level, and access to objects at different sensitivity-levels is controlled strictly by the operating system using MAC controls - one can imagine that a process at a given sensitivity-level to be in a separate 'compartment' from other processes of differing levels. The term 'compartment' is used loosely here to refer to the segmentation that arises as a result of applying MAC controls to labeled objects, and does not refer explicitly to the term used in defining sensitivity-labels. This usage is perhaps closer to what one might intuitively expect when reasoning about isolated groups of processes at different sensitivity levels.

This feature of segmenting data into different sensitivity levels (referred to from now on as compartments) can be used to isolate different processes from each other. For example, a sensitive private-key management agent could be placed in a separate compartment by itself to isolate it from other less sensitive processes.

4.2 Vaulted VPN: Compartmented Stacks

The principle of compartmentalization mentioned earlier yields an architecture for the Vaulted VPN where the processing of IPSec packets is broken into separate user-processes, each running in a separate compartment and handling network packets belonging to the same compartment as themselves.

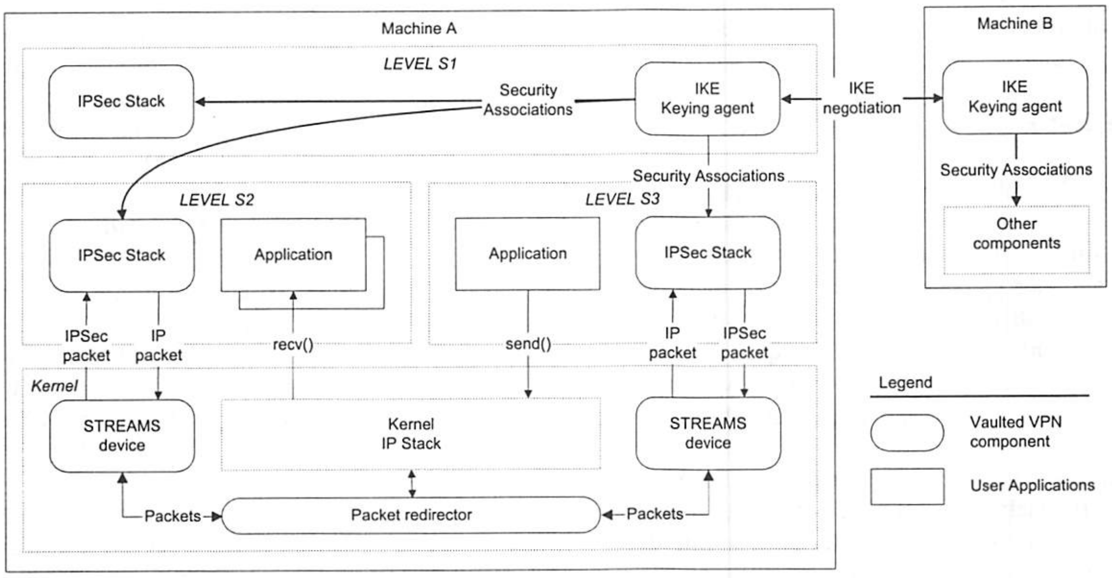


Figure 1 Vaulted VPN architecture showing stacks at different sensitivity levels

The traditional monolithic kernel-level IPSec stack has been changed instead to a series of compartmented, concurrent user-mode processes. In order to make possible the processing of network packets in compartmented user-mode processes, the target platform of the Vaulted VPN is a version of HP-UX 10.24 that has been modified to perform packet redirection by means of a simple kernel-level add-on. The kernel level add-on is necessary as this particular version of the HP-UX operating system uses a BSD-style [10] IP stack that needs to be modified to include packet redirectors.

This architecture (see Figure 1) of the Vaulted VPN consists of:

1. a kernel-level add-on for packet redirection from the kernel to a STREAMS [14] device
2. a series of concurrently executing user-mode IPSec stack-processes running in different compartments. These processes intercept redirected packets from the kernel and perform IP fragment re-assembly and the actual encryption and decryption of IPSec packets [4] [5].
3. a user-mode IKE [1] keying-agent, which performs negotiations with its remote peers in order to derive keying-material in the form of IPSec security-associations for use in encryption and decryption of IPSec packets. The IKE keying-agent performs the negotiations needed to derive the keying-material used in IPSec processing. This keying-material for IPSec security-associations is sent to the various IPSec stacks using a protected message-queue described later.

4.3 Packet Flow

In the Virtual Vault operating system, each packet travelling up or down the network stack is labeled with a sensitivity-label. The kernel has been slightly modified to intercept all incoming and outgoing packets and pass them to a STREAMS-based driver to be queued for further processing. Once queued at the STREAMS-driver, the packets will eventually be read by a user-mode IPSec stack and be processed there.

Each IPSec stack is a separate user process linked to the kernel IP-stack via a STREAMS-based driver. Typically, there is a separate stack for each sensitivity level on the system. Upon startup, the IPSec stacks send a message to the STREAMS-based driver in the kernel to indicate the level that the stack is at. This mechanism of run-time configuration can be abused and many ways exist to counteract this. As an example, one could check the system-supplied user-credentials in the STREAMS-based driver whenever a process attempts to open the driver.

Using this message, the kernel demultiplexes the stream of labeled IP packets to the various user-mode IPSec stacks depending on the value of the label registered previously. This process is purely a simple comparison - the label of each packet has to match the label of a previously registered IPSec stack before the packet can be routed to it. Duplicate IPSec stacks attempting to register using the same label are ignored.

The ability of HP-UX 10.24 to label network packets within the kernel allows the Vaulted VPN to distinguish between packets that are destined for different compartments and to route the packets accordingly. Incoming packets are labeled with the sensitivity-label of the network-interface from which the packet originated. Outgoing packets obtain their labels from the security-attributes of the socket that generated the packet. Because all network packets are labeled internally within the kernel, it is possible to redirect these packets to separate compartments based on their labels and operate on a compartmented basis.

This implies a use of MAC controls that has not yet been described fully. The notion of handling and routing packets on a compartmented basis is a logical extension of the segmented nature of trusted operating systems. Where packets within the kernel used to be processed together regardless of their labels, there now exists some form of segmentation within the network protocol stack of the operating system. This notion can be extended to many other system components, but this report does not explore it any further.

4.4 IPSec Stack - Basic Operation

Each IPSec stack imposes a security-filter on the incoming and outgoing packets. Each time a packet is received from the kernel, the security policy database [2] internal to the IPSec stack is consulted to see if the packet should be processed, dropped or bypass IPSec processing. If no matching entry is found in the policy database, the packet is dropped.

If the policy database specifies that an incoming packet requires IPSec processing, the packet is then inspected to see if IPSec decryption can take place based on the security-descriptors present in the packet. If the decryption processing completes successfully, the decoded packet is sent back to the kernel via the STREAMS-driver to travel up the network protocol stack, usually to a socket's receive-buffer.

For outgoing packets, the security policy database is consulted to see if traffic to the destination specified by the outgoing packet is allowed, and if so, what kind of processing to apply to it. If the outgoing packet is to be IPSec-encrypted, then the relevant encryption-processing is applied and the encrypted packet sent back to the kernel to be subsequently sent out the network interface. Packets banned from transmission by the security policy database are dropped.

4.5 Message Interfaces

The individual IPSec stacks use standard UNIX message-queues to communicate with the IKE keying-agent. These queues are used to trigger IKE negotiations and to pass keying-material from the IKE keying-agent to the individual IPSec stacks. The IPSec stacks also use a kernel-to-user-mode interface in the form of a STREAMS-based driver to send and receive IP packets from the kernel. These interfaces are protected using methods described later.

5 Compartmented Keying Material

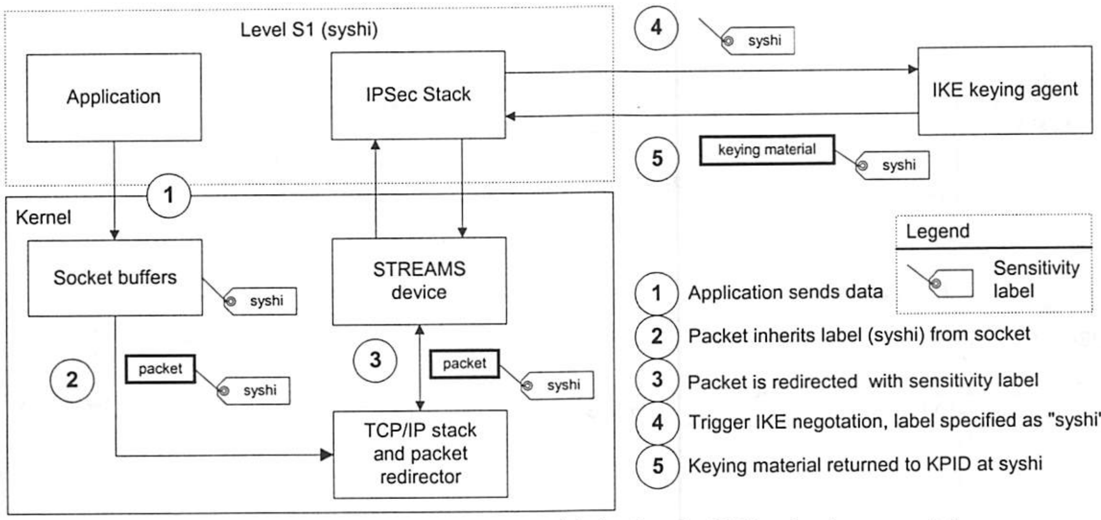
Negotiations to form a new IPSec security-association are normally triggered by an outgoing packet that does not have a matching entry in the security-policy database. In such a case, a message is sent to the IKE keying-agent to trigger a new IKE negotiation, which occurs asynchronously to the processing in the IPSec stacks. The packet in question is queued within the IPSec stack in anticipation of a successful IKE negotiation. Once the IPSec security-association is formed, the queued packets are released to be processed internally by the IPSec stack.

The network packets from IKE negotiations are redirected by the kernel to the IPSec stacks, as it would do for any normal application. However, the Vaulted VPN is configured to behave as a pass-through for IKE packets, so that no special IPSec encryption is performed.

5.1 Labeling of IPSec keying material

The keying material derived as a result of an IKE negotiations is labeled according to the sensitivity-level of the outgoing packet that originally triggered the IKE negotiation. Figure 2 shows how sensitivity-labels of network packets flow through the Vaulted VPN and how they are used to determine the label of derived keying material.

Each IP packet inherits its sensitivity label from the socket that generated it and this label travels with the packet until it is intercepted by the IPSec stack and subsequently used to tag subsequent IKE negotiations.



Level S1 (syshi)
Application
IPSec Stack
IKE keying agent
syshi
keying material
syshi
Kernel
Socket buffers
syshi
STREAMS device
packet
syshi
packet
syshi
TCP/IP stack and packet redirector
Legend
Sensitivity label
1 Application sends data
2 Packet inherits label (syshi) from socket
3 Packet is redirected with sensitivity label
4 Trigger IKE negotation, label specified as "syshi"
5 Keying material returned to KPID at syshi

Figure 2 Determination of sensitivity-level of IPSec keying material

5.2 Labeled Distribution of IPSec keying material

When IKE negotiations are successful, the keying-material derived in the form of IPSec security associations is sent via a private key-channel to the IPSec stack at the same sensitivity-level. The IKE keying-agent does not hold IPSec keying-material beyond the short duration required to derive it and pass it on to the individual stacks.

The net result is that keying-material is segmented according to its sensitivity-label (see Figure 3). The keying-material in any given stack is never duplicated in another, leading to cleanly segmented, non-intersecting subsets of the original keying-material.

5.3 Storage of non-volatile IKE private keys

Private keys used by the IKE keying-agent to digitally sign messages in IKE main-mode negotiations are stored as files and are protected by setting them to the highest sensitivity level, *syshi*. Like the IKE agent, a dedicated compartment would be preferable, but *syshi* is chosen given the static configuration of compartments in HP-UX 10.24. The appropriate permissions-bits are also used to disallow access by everyone else but the VPN pseudo-user. Further measures, including encrypting the keys with a passphrase that is supplied when the Vaulted VPN is started, could be employed but has not been implemented in the current version of the Vaulted VPN.

6 Security Analysis

This section presents a security analysis of various aspects of the operation of the Vaulted VPN. We show how the underlying security-features of a Trusted Operating System are used to isolate and protect the various components of the Vaulted VPN. This is done by examining the system from several aspects – the component processes, the messaging-interfaces between the components and the keying-material travelling between the components. Taken together, this view of security based on the components and their interfaces forms the basis of the security of the Vaulted VPN.

6.1 General security measures

- Each component is run with an effective user-id of a VPN pseudo-user. This user is specifically created for use by Vaulted VPN components only and cannot logon to the system under normal circumstances. This arrangement allows the use of UNIX-style permission-bits to disallow access by non-Vaulted VPN components. The use of system provided file-control databases ensures that only the Vaulted VPN components are assigned this pseudo-id, and allows the use of DAC controls as a more powerful discriminant that would otherwise be possible on standard UNIX systems.

- The *chroot* UNIX-command is used to place each executable component into a restricted filesystem to limit the scope of accessible system files should a security breach occur.

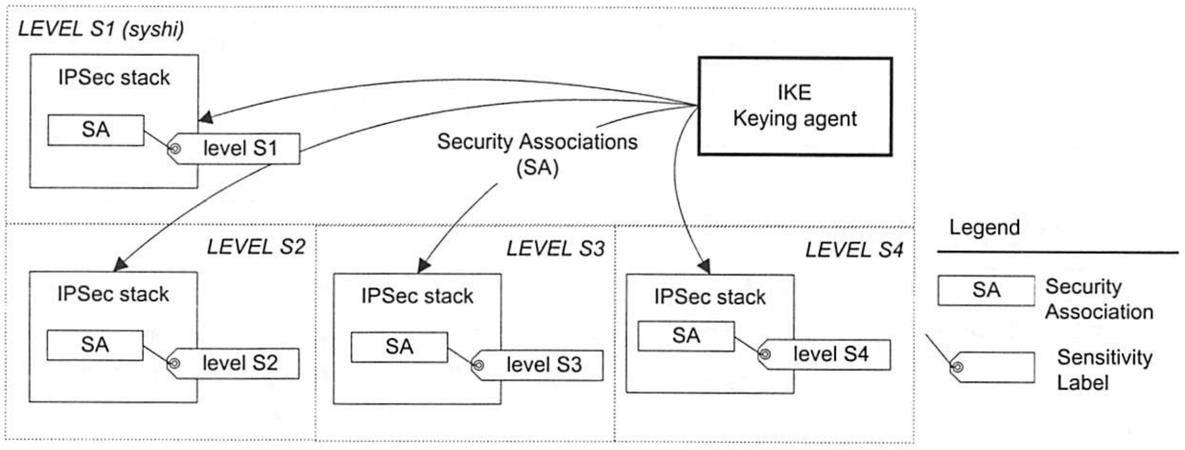


Figure 3 Distribution of labeled IPSec Keying Material

- In other 'full-blown' CMW systems, it is possible to place each component in a separate compartment of its own. However, HP-UX 10.24 offers a static configuration of compartments. Therefore, the safest choice is to place the sensitive components (like the IKE agent) at the highest sensitivity level possible, assuming that this level is not used by very many other processes in a properly configured system.

As a general rule, the IKE keying-agent is placed into the highest-classification possible to isolate it from other processes in compartments of a lower classification. The IKE agent executes with enough privileges to perform certain privileged network operations (such as binding to the IKE-reserved port number), but is otherwise relatively unprivileged.

The IPSec stacks are given almost no special privileges at all because their activities consist almost solely of reading and writing to a device file. Therefore, faults in the Vaulted VPN IPSec stacks do not give root-equivalent access, compared to the situation where a breach in an IPSec stack situated in the kernel allows access to the entire operating system.

6.2 Compartmented Component Processes

The approach taken here is to compartmentalize the various component processes of the Vaulted VPN to isolate them from each other and the rest of the system as much as possible.

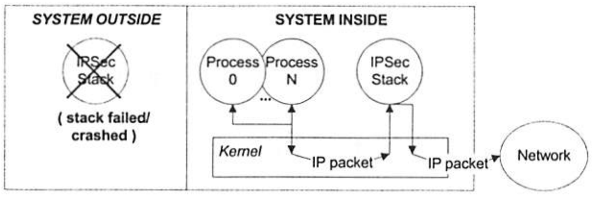


Figure 4 Crash-isolation between compartments

The individual IPSec stacks are also placed into separate compartments, each at different sensitivity levels – typically one stack per compartment. By using this approach, a fault in the IPSec stack in a given compartment does not automatically cause the processing of other stacks to fail (see Figure 4). More importantly, it does not lead to a breach in security for those other stacks.

6.3 Compartmented Keying-material

Keying-material derived from successful IKE negotiations is never kept in a single location with the IKE keying-agent. Instead, it is considered as labeled according to the sensitivity-level of the network packet that initially triggered the negotiation and once derived, is distributed to the appropriate IPSec stacks. The net effect is that the keying-material is only present in the compartment in which it is to be used. This reduces the visibility of keying-material associated with highly sensitive network connections and eliminates the mingling of keying-material found in systems where the entire IPSec stack resides in the kernel.

6.4 Protection of component interfaces

There are two main messaging interfaces in the Vaulted VPN which need to be protected:

1. The keying-channel between the IKE keying-agent and the various IPSec stacks

2. The kernel-to-user-mode STREAMS interface between the kernel IP stack and the various user-mode IPSec stacks

The keying-channel between the IKE keying-agent and the various stacks is a standard message-queue. Protection of this interface is through a combination of Mandatory and Discretionary Access Controls. The IKE keying-agent and stacks are launched under a special pseudo-user. The message-queues are then owned by this pseudo-user by virtue of inheritance and are created with permissions-bits that disallows access by everyone else. This restricts access to the queue to the Vaulted VPN components only, since they are the only components on the system that are configured to use this pseudo-user. In addition, the queue is labeled with the highest classification possible to protect it from access by processes from lower classifications.

The kernel-to-user-mode STREAMS interface used by the IPSec-stack to send and receive IP packets utilizes special device-files that also need to be protected. One device-file is created for each sensitivity level. This ensures that only one interface is accessible at each sensitivity-level. The other interfaces, which serve to handle IP packets of a different sensitivity-level are not visible (and hence, not accessible) from processes running at different sensitivity-levels.

The ownership and permissions-bits on each device-file is set to disallow access to everyone but the special VPN pseudo-user, thereby disallowing access to every other process but the IPSec stack allocated to this compartment. In addition, the kernel-level STREAMS driver checks the system-supplied user-context [14] when the device is opened in order to decide if the device is being opened by a VPN component.

6.5 Preservation of sensitivity-labels across machines

The notion of unauthenticated explicit sensitivity-labels to tag data in labeled IP traffic (e.g. IPSO/RIPSO [12][13], MaxSix [8]) has given way to cryptographic methods such a those used in IPSec. However, it would be desirable if the same net effect could be achieved within the IPSec framework. The solution that IKE provides is the ability to specify security-related attributes in an IKE quick-mode negotiation [15].

During IKE quick-mode negotiations, sensitivity-labels are passed between the negotiating IKE-peers to indicate the level of the incoming and outgoing data. This is stored for every security-association that is formed and is used by the IPSec stack to label processed packets before they move up the network stack. In this way, data that was labeled *syshi* on the originating machine is never "written down" on the receiving end, therefore preserving the sensitivity labels of data crossing the network.

This occurs without necessarily having to pass sensitivity labels in each transmitted IP packet because the sensitivity labels are negotiated first with IKE and stored in the corresponding IPSec security-association used in processing that packet. Packets processed using a particular security association are checked against the label stored with the security association and relabeled (or upgraded) as necessary.

6.6 Differences in security with conventional bump-in-the-stack VPNs

The benefits of this architecture are best understood by comparing it against the design of a conventional bump-in-the-stack IPSec VPN.

- In the Vaulted VPN, the various components and the interfaces between them are protected by the MAC and DAC controls of the Trusted Operating System. A conventional VPN running on a mainstream OS lacks the capability to compartmentalize the IPSec stack, and to isolate keying-material into separate compartments.

- In a conventional VPN, the IPSec stack is usually implemented as a STREAMS-driver or module (e.g. for Solaris) or as a kernel-level NDIS shim as in Windows NT. A fault in the stack (possibly caused by the receipt of a malformed packet) could easily lead to corruption in the kernel and lead the system to crash. In the case of the Vaulted VPN, the IPSec stack has been pulled from the kernel into user-mode and subsequently split into independent, compartmented processes. The kernel is less likely to be affected by a user-mode IPSec-stack crash.

- On a conventional UNIX system, the IKE-daemon has to run as root in order to bind its socket to privileged port number for IKE [16]. Similarly, in Windows NT, the daemon needs Administrator privileges to open a similar port. Following the principles of least-privilege, the keying-agent runs as a relatively unprivileged process. This means that breaking into the IKE keying-agent does not give root-equivalent access.

The last point mentioned above can be important as the IKE protocol is complex and IKE-implementations are still relatively new and lack sufficient code maturity to be considered stable. It is desirable that untrusted IKE daemons not be given root or Administrator access. The various IPSec stacks also execute as relatively unprivileged processes - reducing the scope of any damage from break-ins to the stack.

6.7 Other approaches

The choice of IPSec as the network protocol of the Vaulted VPN contrasts with the approach taken by the HP Praesidium Extranet VPN [17], where the SOCKSv5 protocol [18] is the choice. IPSec-based VPNs have traditionally been seen to be operating at a much lower level than SOCKS-based solutions - which operate at the level of each individual application and have access to higher-level security constructs, including those derived from application-specific domains.

This possible advantage is largely negated by the ability of the kernel-component of the Vaulted VPN to also pass fine-grained kernel security-attributes (derived from the user-processes) to the IPSec stacks. However, SOCKS-based solutions have the advantage of selectively excluding non-security sensitive applications from security-related processing, which allows for reduced overheads.

7 Extensions

This section describes extensions to the configuration of the Vaulted VPN to increase performance and robustness. We show how concurrently executing stacks can de-multiplex the stream of IP packets from the kernel in order to perform IPSec processing in parallel for greater performance. We also describe how the stacks can be dynamically configured to perform load balancing and to run in tandem to create multiple, redundant stacks.

7.1 Concurrent Execution of Stacks

Since the IPSec stacks exist as a series of independent user-mode processes, it is possible to launch as many copies of the stacks as desired. In particular, it is possible to launch two or more stacks to deal with IP packets for a particular compartment (see Figure 5). On platforms that support threads natively, it would be preferably to launch independent threads instead of processes to take advantage of the typically smaller overheads associated with context-switching amongst threads compared to processes.

This can be useful when traffic on a particular system is predominantly of a given sensitivity level. Most mainstream operating-systems feature support for SMP configurations and will distribute compute-intensive processes to idle CPUs, ensuring that speedups are close to being linear. Since the IPSec stacks operate by reading and writing packets via a special device-file tied to a STREAMS-driver [14], it becomes possible to demultiplex the stream of IP packets from the kernel evenly between any number of stacks.

This happens automatically as a result of the default processing at the STREAM-head which is supplied by the platform's STREAMS implementation. The individual stacks themselves are unaware that they are only receiving a fraction of the normal number of packets that they would have received if they were the only stack running at that level. Concurrency control is achieved purely through the operation of the STREAMS-head which is supplied by the platform's STREAMS implementation. The STREAMS-head typically operates at the granularity of individual STREAMS-messages (or IP packets in this case) and handles multiple-readers by queuing their requests internally. A later section lists some preliminary performance figures.

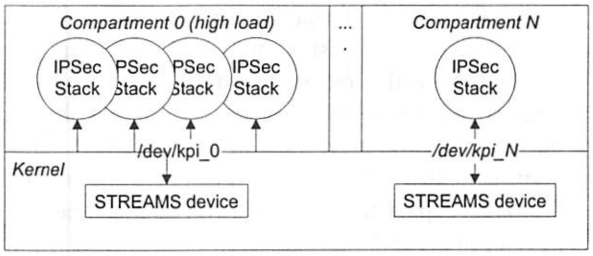


Figure 5 Configuration illustrating a variable number IPSec stacks per compartment

7.2 Multiple Redundant stacks

If it is assumed that IPSec stacks will suffer from the occasional bug, then the possibility exists of sending a corrupted packet to the kernel. To help prevent this, it is possible to implement several versions or implementations of the IPSec stack (possibly using several different languages or compilers).

Because different implementations will typically have different bugs present, the probability of any two (or more) implementations having the same bug under the same conditions is very small indeed (the probability decreases as the number of implementations increase). Therefore, if the multiple redundant implementations are used to "vote" on the output prior to it being accepted by the kernel then corrupt output can be greatly reduced or even virtually eliminated. Such a vote would have to be unanimous, although this approach would not likely solve subtle bugs, or bugs in the original protocol definition itself. As such, this extension although easy to develop, would be of rather less use than one might originally suppose.

7.3 Host-level Accreditation of Sensitivity Levels

It is possible to enforce a scheme where remote hosts are accredited with a range of sensitivity-levels of data that they are allowed to accept. This can be done by encoding the range of sensitivity-levels that a remote host is allowed to accept in extensions to X.509v3 certificates [6] and using these certificates during IKE negotiations.

Figure 6 below shows one such possible encoding for host-accreditation in an X509.v3 certificate. The actual encoding the X509.v3 extension is not described here, although choices could range from schemes using simple ASCII encoding to using table-lookups.

A check can be performed during IKE negotiations to see if the remote host in the negotiations has been accredited with receiving data of the current sensitivity-level of the packet. If the level of the packet falls outside the range specified in the certificate, the IKE negotiation can be programmed to fail. Hence, packets can only travel amongst hosts that have been properly accredited.

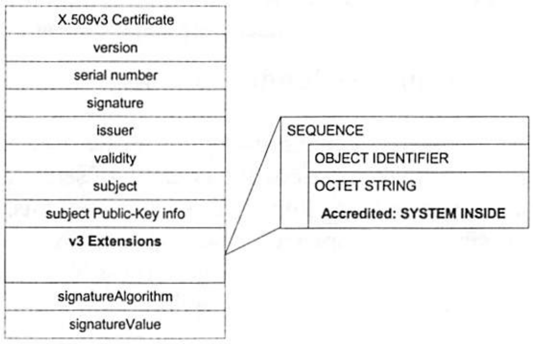


Figure 6 Placement of Host-Accreditation extension in an X509.v3 Certificate

8 Performance Measurements

As expected, the system experiences increased latency (which shows as increased times taken to connect) and lower throughput due to the longer processing pipeline introduced by user-mode stacks and overheads spent in synchronizing access to the STREAMS-device.

8.1 Web-Server Performance

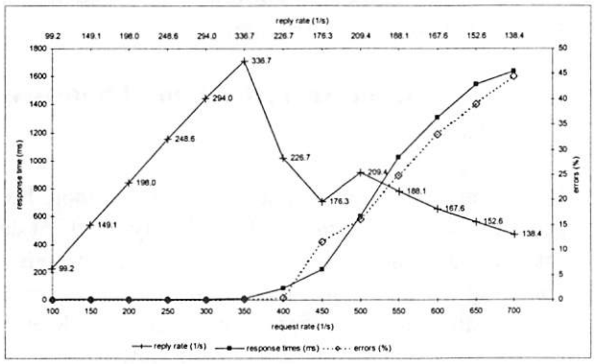


Figure 7 Performance graph-ESP with DES-CBC & HMAC-MD5

Figure 7 above shows the results of tests with *httperf* [22]. The test machine was a HP 9000/871 with 2x160Mhz PA-RISC CPUs with 256MB of RAM running the Apache 1.3.6 Web-server. The test-operation involved repeatedly retrieving a 1-kilobyte file using HTTP over a single TCP connection according to adjustable rates to show peak connection rates, connection-time latencies and errors due to timeouts.

The VPN was configured on the same machine on a 100Mbit/s link to use the IPSec ESP protocol with 8-byte DES-CBC encryption and HMAC-MD5 authentication. The table shows a peak connection rate of around 336 connections/sec after which response-times and errors (as a percentage of total connections) start to increase. This figure is well below what might be expected of this configuration which should be around 700 connections/sec.

The processing associated with DES encryption did not appear to be a significant factor as the figures recorded for a VPN operating in a 'pass-through' mode (i.e. no encryption calculations) were virtually identical as shown in Figure 8 below.

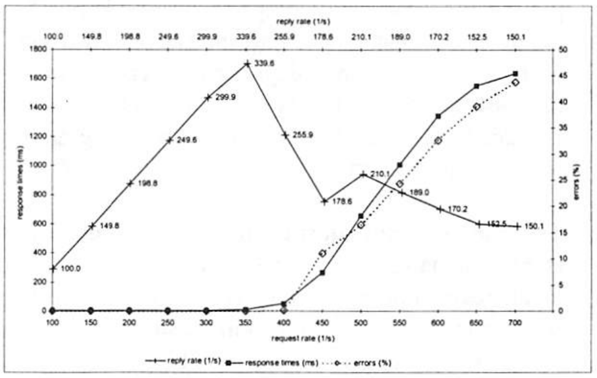


Figure 8 Performance graph – no encryption

8.2 Raw Throughput

Raw network throughput was measured with the *netperf*-tool [23] on a 100Mbit/s Ethernet link. Table 1 and Table 2 below show the results from tests performed on a dual 160Mhz CPU HP 9000/871 server with the VPN configured to use HMAC-MD5 authentication alternately with 8-byte DES-CBC or Triple-DES (3DES) with the IPSec ESP protocol. Tests were performed using both single and dual-stack configurations.

It is thought that the modest speedup on dual-stack configurations is due to the *netperf* client and server competing for CPU cycles with the IPSec stacks. Netperf was observed to consume significant amounts of CPU cycles (>40%) thus depriving the kernel and the IPSec stacks from effective use of the dual-CPU SMP configuration. It was not possible to perform these tests on SMP configurations with more than 2 CPUs due to lack of equipment.

# of stacks	Configuration	Throughput (Mbits/s)
1	DES-CBC & HMAC-MD5	10.0
2	DES-CBC & HMAC-MD5	11.5
1	3DES & HMAC-MD5	6.79
2	3DES & HMAC-MD5	9.25

Table 1 Netperf using unidirectional TCP_STREAM

# of stacks	Configuration	Throughput (Mbits/s)
1	DES-CBC & HMAC-MD5	13.1
2	DES-CBC & HMAC-MD5	13.8
1	3DES & HMAC-MD5	7.9
2	3DES & HMAC-MD5	8.0

Table 2 Netperf using unidirectional UDP_STREAM

The issue illustrated by running *netperf* in dual-stack configurations is one of resource-management. The run-time behaviour of user-mode applications impact directly on the performance and behaviour of the VPN because the system described does not offer adequate control over resource consumption, namely CPU-usage.

As noted in [21], conventional OSes (such as the system described here) currently treats individual processes as resource principals while failing to properly account for the kernel resources that they consume. The simple task of bandwidth-restriction is complicated by the need to calculate the accumulated CPU-usage of a packet as it flows from an interrupt-service routine in the kernel up to a user-mode handler and then down again. In such a scenario, one must view the IPSec stacks as extensions to the kernel and attribute the resources they consume to the originating resource principals.

To increase performance, the IPSec stacks could be restructured as a pool of kernel-level threads. However, the user-space design can be improved somewhat – perhaps by increasing the efficiency of event-driven servers (as each IPSec stack invokes *select*() repeatedly) as described in [24], or by using platform-dependent APIs which perform memory-transfers direct to-and-from user-space instead of the STREAMS interface currently being used.

9 Summary

Conventional bump-in-the-stack VPNs have well-known security hotspots such as running the IKE keying agent as root, or placing IPSec processing in the kernel which exposes the kernel to faults in the IPSec stack. A solution to some of these problems is to remap the design of a conventional IPSec VPN to a trusted operating system (such as HP-UX 10.24) to take advantage of the security-features present in such a system. Through the use of a trusted system, further refinements become possible – the IPSec stack and the associated keying-material can be compartmented to offer an even greater degree of security and robustness.

The Vaulted VPN architecture features multiple asynchronous user-mode IPSec stacks which execute concurrently and independently from each other. In HP-UX 10.24, these stacks are placed in separate compartments and handle only the IP packets that match the compartment that they're in. IKE keying-material is distributed to the relevant compartments and not kept in a single location. Sensitivity-labels of data crossing the network is preserved through the use of IKE negotiations without requiring the label to be transmitted in each encrypted IPSec packet.

The IKE keying-agent is never run as root to prevent faults in the agent from giving root-access. Because the relatively unprivileged IPSec stacks are placed in separate compartments, a failure in one does not necessarily lead to failures or security breaches in another. The Vaulted VPN IPSec stacks are compartmented, stripped of most privileges and do not possess a complete view of all the keying material on the system. The various messaging channels between Vaulted VPN components are protected by a combination of MAC and DAC to offer a higher level of security than would otherwise be present on a non-MultiLevel Secure operating system.

Because the IPSec stacks are user-mode processes they can be configured dynamically to execute in parallel for better performance and load balancing. They can also be configured to form multiple redundant stacks for checking the correctness of processed IPSec packets. Host-level accreditation in the style of IPSO/MaxSIX can be achieved by reinterpreting the original mechanisms and performing any needed authentication in the IKE negotiation phase. The failure or success of forming a security-association is directly tied to IKE negotiations that can be expanded to encompass other security-models extant.

References

[1] The Internet Key Exchange (IKE) protocol (RFC 2409)- D. Harkins, D. Carrel, Nov 1998

[2] Security Architecture for the Internet Protocol (RFC 2401) – S. Kent, R. Atkinson, November 1998

[3] IP Security Document Roadmap (RFC 2411) – R. Thayer, N. Doraswamy, R. Glenn, November 1998

[4] IP Encapsulating Security Payload (RFC 2406) – S. Kent, R. Atkinson, November 1998

[5] IP Authentication Header (RFC 2402) – S. Kent, R. Atkinson, November 1998

[6] Applying Military Grade Security to the Internet, Computer Networks and ISDN Systems, Volume 29 Number 15 – C. I. Dalton, J. F. Griffin, November 1997

[7] Internet X.509 Public Key Infrastructure Certificate & CRL Profile (RFC 2459)- R. Housley, W. Ford, W. Polk, D. Solo, January 1999

[8] HP-UX CMW MaxSix Administrator's Guide, Hewlett-Packard Co., 1995

[9] Compartmented Mode Workstation Evaluation Criteria, Report DDS-2600-6243-91, Defense Intelligence Agency, 1991

[10] TCP/IP Illustrated Vols 1, 2 and 3– Gary R. Wright, W. Richard Stevens, Addison-Wesley 1995.

[11] Secure Computer Systems: unified Exposition and Multics Interpretation, MITRE Technical Report MTR –1997 MITRE Corporation NTIS AD A023588/7– D. E. Bell, L. J. LaPadula March 1975

[12] U.S. Department of Defense security options for the Internet Protocol. (RFC1108) – S.T. Kent, November 1991

[13] Revised IP Security Options (RFC 1038) – M. St. Johns, January 1988

[14] STREAMS/UX for the HP 9000 Reference Manual, Hewlett-Packard Co, 1995

[15] The Internet IP Security Domain of Interpretation for ISAKMP (RFC 2407), D. Piper, November 1998

[16] IANA Assigned Port Numbers, ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers

[17] HP Praesidium Extranet VPN, Hewlett-Packard Co., 1998

[18] SOCKS Protocol Version 5 (RFC 1928), M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones, March 1996

[19] CERT Advisory CA-97.28 IP Denial-of-Service Attacks (Teardrop_Land), CERT Coordination Center, May 1998.

[20] CERT Advisory CA-96.26 Denial-of-Service Attack via PING, CERT Coordination Center, December 1997

[21] Resource containers: A new facility for resource management in server systems, G. Banga, P. Druschel, J. C. Mogul, USENIX Security Symposium 1997

[22] httperf – A Tool for Measuring Web Server Performance, D. Mosberger, Tai Jin, Hewlett-Packard Research Labs

[23] netperf 2.1– Network Performance Benchmark, http://www.netperf.org/

[24] Better Operating Systems Features for Faster Network Servers, G. Banga, P. Druschel, J. C. Mogul

Enforcing Well-formed and Partially-formed Transactions for Unix

Dean Povey
Security Unit
Cooperative Research Centre for Enterprise Distributed Systems
Queensland University of Technology, Brisbane, 4001
povey@dstc.edu.au

Abstract

While security is a critical component of information systems, at times it can be frustrating for end users. Security systems exist to minimise the risks of allowing users to access and modify data, but rarely do they consider the risks of *not* granting access.

This paper describes an access control system which is *optimistic*, i.e. it assumes accesses are legitimate, and allows audit and recovery of the system when they are not. The concepts of *well-formed* and *partially-formed* transactions as mechanisms for constraining pessimistic and optimistic accesses is briefly described, and the paper details a prototype implementation for the Solaris operating system which provides a reference monitor for enforcement of both these transactions.

1 Introduction

One of the main objectives of security systems is the management of risks by controlling access to resources. These systems implement security policies which seek to enforce the principles of *least privilege* and *separation of duties* to ensure that users are given the minimum amount of privilege necessary to perform tasks.

However, what most systems often do not consider is the risk of *not* granting privileges. Either due to unforeseen circumstances, or because the access control system is not flexible enough to support complex policies, enforcing restrictive access control can impede a user's legitimate activities.

This paper considers security from the perspective of reducing risks in an environment where access control is performed optimistically. In such a system, access is assumed to be legitimate, and granted automatically. Integrity is ensured by verifying the validity of actions after the fact, and providing a way of rolling back user actions when they are deemed inappropriate. The paper categorises user actions as being one of three types: legitimate, questionable and dangerous, and shows how both pessimistic and optimistic access control mechanisms are useful for constraining different classes of action.

The notion of *partially-formed transactions* (an extension of Clark and Wilson's *well-formed transaction*[5]) is described as a mechanism for ensuring integrity in an optimistic access control system and this concept is presented within a formal context.

The paper also describes the design and implementation of `tudo`, a prototype for the Unix operating system that enforces both well-formed and partially-formed transactions. `tudo` provides a reference monitor which controls access to system calls using the system call tracing features of the Solaris `/proc` filesystem, and provides detailed logging of actions and recovery from filesystem modifications. The security and limitations of the prototype are evaluated and some future directions for the research are discussed.

2 Motivation

In designing access control policies for an organisation, security administrators need to balance the needs of users with the need to maintain system integrity. However, these two requirements are often conflicting. In general, it is most efficient for users to have minimal barriers to completing tasks, particularly in environments which are collaborative and time-critical. Maintaining system integrity however, requires that a user's access privileges be restricted to a minimal set, to ensure that the amount of damage that can be caused (accidental or otherwise) is minimised.

The situation is further exacerbated in environments such as Unix, where the simple access control scheme supported by the system is often not flexible enough to support complex security policies. For example, in Unix, it is not possible to authorise the following actions without giving the user additional privileges:

- allow append-only access to a file;
- bind a network socket to a port < 1024;
- allow read/write access to a raw disk device; or
- allow mounting or unmounting of a filesystem.

Furthermore, security systems usually enforce a policy appropriate for the normal operation of a system, but do not allow flexibility in dealing with disaster scenarios or dynamically changing environments.

For example, suppose a user with a critical deadline turns up early to work to find that the line printer daemon configuration has been changed, and they are unable to print their project proposal. As the system administrator is nowhere to be found, their only option is to attempt to fix the problem themselves. In normal circumstances, allowing the user to alter the line printer configuration file would be unacceptable, however in this situation, not allowing them to alter it will mean that they cannot meet their deadline.

2.1 Categories of user actions

When considering how a given access control mechanism implements its security policy, we can think of the set of possible actions a user might wish to perform as being divided into three separate categories:

Legitimate Actions that are explicitly allowed by a system's access control mechanism.

Questionable Actions which may represent legitimate user behaviour, but require access privileges not normally given to the user.

Dangerous Actions that are explicitly disallowed by the security mechanism, and which represent accidental damage or malicious intent by users.

Security systems only allow legitimate actions. However, either due to unforeseen circumstances, or because the access control system is not sufficient to support the complexity of authorisations necessary for a particular task, a user will occasionally need to perform actions which will be deemed questionable. A user who needs to perform such actions has two options:

1. convince a benevolent system administrator to perform the action for them; or
2. perform the action through a trusted entry point provided by the administrator.

2.1.1 Questionable actions through a system administrator

Most operating systems (including Unix) have the concept of a privileged user or superuser who can override the security mechanisms of the system to perform a questionable action. However, there are often occasions (such as the example above) when the system administrator cannot be found (or a *benevolent* system administrator cannot be found), or where they are too busy to complete the request in a timely fashion. In addition, requiring all questionable actions to be approved by the system administrator is inefficient, particularly when the action is commonly performed.

2.1.2 Questionable actions through trusted entry points

Unix supports the notion of trusted entry points, by allowing a program to be executed with setuid or setgid permissions. When executed, the program is run with the privileges of either the owner or group of the executable file (depending on which permission is set). The main benefit of this approach is it allows the administrator to authorise questionable actions which are regularly executed, and known to be trusted. However, in many cases, the privileges required to perform the operation will demand that the program be executed as the superuser. Because executing with the superuser privilege provides full access to the system, these programs then often become a target for misuse by exploitation of weaknesses in their design and implementation (for examples of these exploits see [12] and [15]). The existence of such widespread exploits make administrators reticent to provide too many entry points for users to execute questionable actions, as they do not always have the time or expertise to determine if the setuid/setgid program has been implemented securely.

Another approach is the use of utilities such as `sudo`[19] and `super`[20], that allow the execution of specified programs with superuser privileges. The advantages of such utilities over normal setuid/setgid programs are that they provide richer access control semantics using a high level policy language, and they sanitise the environment in which the specified programs run to avoid known attacks. This simplifies the process of certifying a particular program as being secure, as many vulnerabilities are averted by the `sudo` and `super` utilities themselves.

However, such approaches are still susceptible if the programs they execute behave in unexpected ways or are vulnerable to data dependent attacks. The `sudo` and `super` utilities provide a safer way to access trusted entry points, but once the entry point has been authorised, the program being run has carte blanche access to the system.

Lastly, mechanisms which support trusted entry points are only useful for cases where the need for a questionable action is known in advance. In the example given above, it is unlikely that an administrator will to provide a entry point to allow users to modify the printer configuration file, as such access is probably legitimate only as a one-off occurrence.

2.2 Optimistic access control

Another way of approaching the issue of authorising questionable access, is to decide whether authorisation should be pessimistic or optimistic. In the schemes described above, authorisation is pessimistic, i.e. it is assumed that all accesses are potentially dangerous, and only those cases explicitly authorised by the administrator are permitted.

An optimistic approach, is to assume that most accesses users will request will be legitimate, and any access that does not fall in the dangerous category should be granted. The actions are audited so that if a system administrator later decides that an access was unreasonable, they can take corrective action to repair any damage. Because the scheme is optimistic it assumes that such instances will be rare. For example, in the situation above where a user needed to edit the printer configuration file to fix a problem, the system administrator should be able to determine that under the circumstances, such action was legitimate. In the case that a user acted inappropriately in performing such an action, the administrator can take steps to make sure that the user is aware of their mistake, take punitive action, or specifically disallow them that privilege in future.

In an optimistic system, it is the responsibility of users to ensure that they are acting in accordance with the organisations policy, and it is the responsibility of system administrators to enforce the policy. The purpose of the access control system is simply to prevent dangerous accesses, and to provide accountability and auditability of user actions. This assumes that in general, users can be trusted to behave appropriately, that the need for users to execute questionable actions will only arise infrequently, and an administrator can recover from actions which damage the system integrity.

However, clearly such a mechanism is open to abuse if no additional constraints are placed on the actions which the user performs. If users are allowed open slather on the system, then the potential for both accidental and intentional misuse can cause serious risks.

One way to ensure that integrity is maintained in an optimistic system, is to ensure that any action that the user performs can be rolled back or compensated. This allows the system administrator to decide after-the-fact that an action was unreasonable and recover the system to a valid state. While, the cost of recovery might be high, if the instances requiring recovery are relatively infrequent, this cost will be balanced by the increased empowerment of users.

2.3 Summary

Balancing a security policy between the needs of users, and a need to secure the system is a difficult task. Security administrators need to consider not only the appropriate authorisations for the general running of the system, but what is appropriate in unforeseen circumstances. In addition, administrators may need to contend with an access control system that is not sufficient to support complex authorisation semantics, and need to consider how to give controlled access to certain resources which exceed a user's normal privileges.

While the Unix operating system currently provides some limited support for the later, it provides no optimistic method by which users can gain privileges in unforeseen circumstances. Thus, a mechanism is needed to provide better support for both legitimate and questionable accesses for Unix users.

3 Formal Model

3.1 Clark-Wilson Integrity Model

In their seminal paper [5], Clark and Wilson argued that unlike military systems whose main aim is to prevent disclosure, the goal of commercial security systems is to ensure that the integrity of data is preserved. They define the concept of a *well-formed transaction* as a transaction where the user is unable manipulate data arbitrarily, but only in constrained ways that preserve or ensure the integrity of the data. A security system in which transactions are well-formed ensures that only legitimate actions can be executed.

Clark and Wilson's paper presents a formal model for data integrity which consists of a number of components:

Constrained Data Items (CDIs) Objects that the integrity model is applied to.

Unconstrained Data Items (UDIs) Objects that are not covered by the integrity policy (eg. information typed by the user on the keyboard).

Integrity Verification Procedures (IVPs) Procedures for verifying that CDIs conform to the integrity policy.

Transformation Procedures (TPs) A procedure which transforms a CDI from one valid state to the other. This is the Clark-Wilson concept of a well-formed transaction.

Clark and Wilson's paper presented nine rules for the enforcement of system integrity which are listed in Figure 1.

Rule	Description
C1	IVPs must ensure that CDIs are valid
C2	TPs on CDIs must result in a valid CDI
C3	Separation of Privilege & least privilege
C4	TPs must be logged
C5	TPs on UDIs must result in a valid CDI
E1	Only certified TPs can operate on CDIs
E2	Users must only access CDIs through TPs for which they are authorised
E3	Users must be authenticated
E4	Only administrator can specify TP authorisations (i.e. MAC)

Figure 1: Clark-Wilson Integrity Rules

The rules are divided into two types: certification and enforcement. Both involve ensuring compliance with the integrity policy. But certification involves the evaluation of transactions by an administrator, whereas enforcement is performed by the system.

Examination of the Clark-Wilson model, shows that the nine rules seek to enforce four separate, but related security properties:

Integrity An assurance that CDIs can only be modified in constrained ways to produce valid CDIs. This property is ensured by the rules: C1, C2, C5, E1 and E4.

Access control The ability to control access to resources. This is supported by the rules: C3, E2 and E3.

Auditability The ability to ascertain the changes made to CDIs and ensure that the system is in a valid state. This is ensured by the rules C1 and C4. However, in the Clark-Wilson model, log data is modelled as a CDI, which means the rules for integrity also apply.

Accountability The ability to uniquely associate users with their actions. This requires authentication of such users which is enforced by rule E3.

3.2 Partially-Formed Transactions

While well-formed transactions provide integrity assurances for legitimate actions, they do not allow for the possibility of questionable actions. As indicated in section 2.2, such actions require a relaxation of some constraints, and a mechanism which allows recovery of the system, in the event of failure.

The concept of a *partially-formed transaction*[16] can be used to describe transactions where the integrity of the data is not guaranteed, but where a compensating transaction exists to return data to a valid state. The transaction is said to be partially-formed, as the integrity of the system is only guaranteed by the compensating transaction, and not by constraining the actual action itself.

To support partially-formed transactions, a set of rules is needed to describe how such transactions can be constrained. Like those for well-formed transactions, these rules must ensure the integrity, access control, auditability and accountability of the transaction is maintained. These rules are summarised below.

C1 IVPs must ensure that all CDIs are in a valid state at the time the IVP is run.

C2 All TPs must be certified to provide a compensating TP which will return any modified CDI to a valid state.

E1 The system must ensure that only TPs which have been certified against requirement C2 are allowed to run.

E2 The system must ensure that users can only use those TPs for which they have been authorised.

E3 The system must authenticate the identity of each user.

E4 Each TP must write to an append-only log all the information required to reconstruct the operation, along with the corresponding compensating TP to reverse it.

E5 Only an administrator is permitted to authorise users to access TPs (Mandatory Access Control).

Rule C1 is needed (as in the Clark-Wilson model) to determine whether or not a CDI is in a valid state in order to satisfy the auditability requirement. Rule C2 certifies that the TP corresponds to notion of a partially-formed transaction. Rule E1 exists so that only TPs which can be recovered from are allowed to be executed, and rules E2 and E3 provide the accountability and access control requirements. Rule E4, which provides the accountability requirement, is specified as an enforcement rule rather than a certification rule (contrary to the Clark-Wilson model). This is needed, as it is not possible to model the log as a CDI in the partially-formed transaction model because recoverability relies on this logging information to be kept secured.

3.3 Compensating actions

One issue is whether compensating actions themselves should be well-formed or partially-formed. If they are well-formed, then they will consider the data item being recovered to be a UDI which must be converted to a CDI. Having compensating actions which are well-formed mean that the system may always be rolled back to a valid state.

However, if compensating actions are only partially-formed, this provides the ability to support undo-redo semantics. It may be that the compensating actions themselves are questionable, and a system administrator may wish to undo the recovery. There are benefits to both ideas, and as such the question should be left open for the individual system designer to decide.

3.4 Fit-for-purpose

Whether partially or well-formed transactions should be used depends on the particular environment we are trying to secure. In the case of the accounting or financial systems on which Clark and Wilson's model was based, the strong integrity assurance obtained by using well-formed transactions are needed, as the benefits of providing an optimistic access control system are outweighed by the risks of fraud. However, in the case of highly collaborative and time-critical environments (e.g. health care), it may be more important to support the optimistic system, and use the compensating transactions to clean up the mess later.

In practice, it is likely that both types of transaction will co-exist. Security administrators may wish to enforce a general integrity policy using well-formed transactions, but provide entry points for a subset of partially-formed transactions to allow for unforeseen circumstances. Execution of partially-formed transactions can then be carefully monitored to ensure that they are only used in appropriate circumstances.

Partially-formed transactions may also be useful in the case where the integrity of a TP cannot be easily certified, but the ability to reverse its effects on the system can be. For example, an untrusted application which is allowed write access to the filesystem can be allowed to run as a partially-formed transaction providing the system is able to detect the changes made to files and provide a way to reverse them.

3.5 Summary

The Clark-Wilson integrity model provides a means by which a system can be constrained to ensure that only legitimate accesses can be executed. However, for the reasons given in section 2, it appears that there is a need to provide a controlled way for relaxing these restrictions to support user's abilities to perform their work. The notion of a partially-formed transaction provides a mechanism by which a system can seek to be optimistic about authorising questionable user actions.

4 Design

It appears on even the most cursory examination of the Unix operating system that it is incapable of enforcing partially-formed transactions, and that many standard Unix applications have not been certified to provide the logging requirement necessary for well-formed transactions. In the following sections, the design and implementation of a prototype is described which enforces these rules for the Unix operating system.

In designing a secure mechanism for enforcing well-formed and partially formed transactions, a number of issues had to be considered:

Security This is the first and foremost requirement for the system. The design must ensure that the rules for well-formed and partially-formed transactions are adhered to enforced, and that the mechanism does not create any other vulnerabilities for the system. Standard good programming practices, such as those described in [9] and [4] need to be adhered to, and the prototype should reflect basic, secure design principles such as those described in [18].

Flexibility The system must provide an expressive policy mechanism which enables administrators to configure the mechanism for a range of applications.

Simplicity The design and implementation should be as simple as possible to ensure that it can be easily understood and evaluated. In addition, any policy mechanism must be easy to use and understand.

Extensibility The design should be modular and easy to extend.

4.1 Reference Monitor

The design of the secure mechanism is based on the reference monitor concept described in [21]. A reference monitor is "an abstract machine that mediates all accesses to objects by subjects" [14]. Figure 2 shows how this reference monitor was implemented.

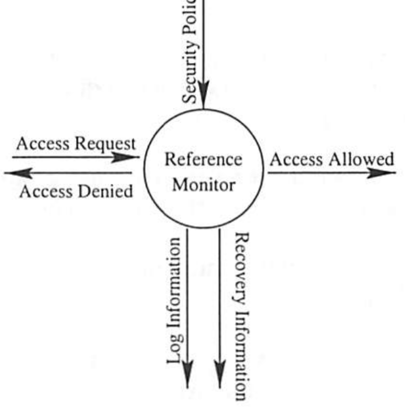


Figure 2: Reference Monitor

The reference monitor takes as input a security policy expressed in a high level language, and a particular action which is being performed. It first ensures that the user is authenticated, then evaluates the security policy to see whether the action complies. If the action is accepted, then recovery information may be stored to roll back the action at a later date. If it is rejected, the access is aborted and an error returned In either case, information about the access may be appended to a log file. The security of the reference monitor is critical as it will be solely responsible for ensuring that the rules specified in section 3 are enforced.

4.2 Logging and Recovery

Well-formed and partially-formed transactions require the accumulation of log information for audit and recovery purposes. This log information must be kept secured, and the implementation should provide a mechanism by which the level of logging and recovery can be specified. The implementation should also provide a mechanism for sending alerts for dangerous accesses, either by email or logging to the system console.

4.3 Policy Language

Designing a system for enforcing both well-formed and partially-formed transactions requires building a mechanism which not only enforces the necessary rules, but provides a rich way of expressing the policy mechanisms. As discussed in section 2, one of the problems with the Unix operating system is that its access control mechanism is too simplistic to cope with complex access control policies, and therefore some user actions may be classified as questionable by the system, which would be declared legitimate in a more flexible system.

The aim of the policy language is to be simple yet flexible enough to support rich access control paradigms such as Role-Based Access Control and Domain and Type Enforcement (see sections 5.5.2 and 5.5.3).The policy language should also support the grouping of related actions into *tasks* to provide a simple mechanism for specifying authorisations based on a number of finer grained actions.

In addition the policy language should be flexible enough to support the addition of future actions or constructs.

4.4 Types of actions supported

In the initial prototype, only control of filesystem actions will be supported. While this is appropriate for an initial proof-of-concept, in a robust system other actions such as networking and interprocess communication should be controlled. Actions on which authorisations can be based should include standard read, write and execute privileges, along with append-only, delete, truncate, and changing of file status information.

5 Implementation

5.1 Overview

`tudo` (Trusted User Do) is an application designed to enforce both well-formed and partially-formed transactions in a Unix operating system. It is based on `sudo`[19], originally developed at SUNY-Buffalo and since enhanced by numerous contributors. `sudo` provides a way to give users restricted root access by allowing them to execute certain programs with superuser privileges. `tudo` builds on this idea to support fine-grained access control of files and directories, and to provide the logging and recovery features necessary to support well-formed and partially-formed transactions. Despite being based on `sudo`, `tudo` has been written completely from scratch to provide these features.

`tudo` has been implemented on the Solaris operating system, although it has been designed to be portable to other Unix variants. The approach used to provide a reference monitor is similar to that taken in Janus (an environment for untrusted applications) [11]. `tudo` uses the system call tracing facilities provided by the `/proc` filesystem in the Solaris operating system to trap access to certain system calls and ensure that this access complies with the specified security policy. `tudo` forks a child to perform the requested action, and traces this process and its descendants to ensure they comply with the policy. Unlike Janus however, `tudo` allows recovery from certain actions by logging information to enable recovery in the case of failures.

tudo runs as a setuid root application. Actions are invoked, either by specifying command line arguments to execute, or giving a task or role the user wishes to perform (see below for a more detailed explanation of these constructs). When tudo is started, it first prompts the user for their password and then checks to see if the requested action is authorised. For example, a user wishing to edit the line printer configuration file might invoke:

```
$ tudo vi /etc/lpd.conf
Password:________
...
"/etc/lpd.conf" 99 lines, 2946 characters
```

Then restart the printer daemon using the lpc restart command (a normal setuid application).

tudo also provides the ability to group a number of related actions together in a *task*. Tasks provide controlled entry points to privileges, as the task can specify that a specific command and arguments must be run. Tasks can also include other tasks to enable new authorisations to be constructed from smaller building blocks. For example, an administrator might construct a reboot task to allow users to reboot the system. Authorised users could then invoke the shutdown by referring to the task rather than the command line arguments: e.g

```
$ tudo -t reboot
Password:________

Shutdown started. Thursday March  4 19:30...
```

Tasks can also be dynamically invoked, when tudo executes a command for which a particular task is defined. When a task is dynamically invoked, it is run with the privileges specified for the task, rather than those specified for the user.

Lastly, tudo provides the ability to specify *roles* to associate principals and authorisations. A role includes a list of the tasks and authorisation rules specific to that role, along with a list of users, groups, and other rules which may assume it. A user may invoke a shell as a given role using the -r flag to tudo. For example, a role might be created to allow its members to remove temporary files if the filesystem is filling up.

```
$ tudo -r SpaceCleaner
Password:________

# find . -name core -print | xargs rm
# find /home -name '*~' -print | xargs rm
# rm -rf /home/*/.netscape/cache/
```

5.2 Access Control

When a system call trap occurs, tudo consults its list of access rules to determine if the access is allowed. If it is, tudo does any necessary log and recovery processing and tells the process to continue executing. If the action is not allowed, the process is directed to abort which causes the system call to be interrupted with an EINTR errno. Because some applications attempt to retry system calls which fail with this error, tudo supports a feature from the Janus system which detects a number of failed system calls with the same parameters and then kills the process.

Because a possible race condition may exist, tudo also checks the result of the system call in some circumstances (e.g. opening a file), to ensure that the state of the object being accessed hasn't changed between the access control decision and execution of the system call. If tudo finds that the object has changed, it immediately kills the process by sending it a SIGKILL signal and sends a message to the log. tudo also checks the result of system calls when recovery and logging is turned on to determine if the action was successful or not.

5.3 Secure Logical Partition

Because the integrity of the configuration, log and recovery information is crucial to the security of the system, tudo maintains a *secure logical partition* where these files are kept. This is a logically separate portion of the filespace which has been set aside specifically for that purpose. Access to the secure partition is automatically denied by implicit rules unless the user has specific authorisation as a tudo administrator.

5.4 Recovery

tudo provides a very simple recovery feature which allows certain modifications to the file system to be rolled back. tudo allows the administrator to specify whether or not certain actions can be rolled back, and can specify what the default policy is for particular tasks, roles or for all users. tudo only allows recovery from an entire session (i.e. the effect on the system from when tudo is invoked to when the last child it is tracing exits). This simplifies the process of recovery, as tudo does not need to worry about serialising the order of events which occur within the session[1]. tudo compares the states of accessed files and directories at the start and end of a unit of work and logs enough information to reverse any changes made. tudo provides utilities to examine the event log, and roll back individual actions. Rollback in tudo is implemented in the following manner:

- When an `open()` system call is invoked, tudo first checks to see if the access is allowed (depending on the flags). If the argument to `open()` is a regular file or a directory, and is being modified, tudo then makes a backup copy of the file on the secure partition. When the session completes, tudo creates a context diff of the modified file using the `diff` command. This context diff can then later be used as input to the `patch` command to restore the file. In the case that the modified file is a binary file, the original backup is kept. In either case, if tudo detects that the file has not changed then it logs this fact and removes the backup copy.

- A `creat()` system call, or an `open()` with `O_CREAT` which creates a new file is rolled back by deleting the created file.

- A `rename()` call is rolled back by changing the old filename back to the new filename.

- When a `chmod()`, `chgrp()` or `chown()` is performed on a file or directory, tudo first calls `stat()` on the file and saves the old file permission/ownership information. Rollback is performed by restoring the permissions to their original state.

- When a file is deleted using the `unlink()` call, tudo makes a copy of the file on the secure partition. However, often administrative task involve removal of large amounts of unnecessary files (e.g. rm /tmp/*), so setting a recovery option for removals can result in a large amount of diskspace being consumed. For this reason, recovery of unlinked files should be avoided. Where possible, administrators should rely on backups to provide recovery for removed files. It should be noted that tudo will not backup the file if there is a hard link to the file somewhere else on the filesystem, but instead logs the inode of the link.

[1]However the issue of serialising the order of multiple tudo sessions is not considered (see section 6.4.1

If a tudo session exits unexpectedly without cleaning up the recovery information this will be detected the next time tudo is invoked, and steps will be taken to clean up the session.

A recovery action runs as a normal tudo action, so this also provides redo semantics. However, it should be noted that users will not always have the necessary privileges to rollback an action, so this may have to be performed by an administrator.

5.5 TUPL

5.5.1 Overview

tudo uses a simple language called TUPL (Trusted User Policy Language) to express access control policies. TUPL supports several constructs which allow the easy expression of policies to support well-formed and partially-formed transactions. TUPL allows the administrator to define access rules which associate a collection of users, groups or roles with an action. These rules take the form:

```
allow <action> <args> [by <principal>]
    [ "[" options "]" ]
deny <action> <args> [by <principal>]
    [ "[" options "]" ]
```

The `principal` may specify a user, a group or a role, or a list of these. An `<action>` may be a list of tasks, predefined actions or the special action `any`, which means that the access applies to any action. Figure 5.5.1 lists the predefined actions which are valid. The design of the policy module is such that more actions can be easily added.

Target	Actions
files	read, write, modify, append, create, trunc, rm, exec, rename, link, chmod, chown, chgrp
dirs	ls, chdir, rmdir, chgrp, chown, chmod

In addition, TUPL allows a number of constructs to be specified which can be combined with the access rules to implement more complex policies. TUPL supports the following constructs:

`type` Specifies a class of objects for which particular rules can be enforced. Currently only the `file` and `dir` classes are supported, although the policy module is designed so that new class types can be added. e.g. all files in the CVS directory owned by the engineering or research group can be specified using:

```
type project_code {
    class: file;
    path: "/usr/local/cvs/*";
    owner: (engineering, research);
}
```

`task` Identifies a particular task type which might be allowed, and the actions which should be allowed to be performed when executing that task. The task has an optional field `command`. When `tudo` is executed with the `-t` option, and a task is specified, the command and arguments in this field will be executed, or in the case that none is present, `tudo` will fork a shell. If a task is not specified in this way, then it may be dynamically invoked when `tudo` encounters a request to execute a program which matches the `command` field. e.g. a task which allows users to edit the line printer configuration files can be specified using the following construct:

```
task edit_lpd_conf {
  command: "vi /etc/lpd.conf";
  description: "Edit lpd.conf";
  rules: {
    //Allow user to run vi
    allow exec "/bin/vi";

    //Allow vi to open shared libs
    //Assumes lib_path_type already
    //defined
    allow read lib_path_type [log=0];

    //Allow read/write on tmp files
    allow all "/var/tmp/*" [log=0,
       recover=no];

    //Allow read/write on lpd.conf
    allow (read, write)
        "/etc/lpd.conf"
        [log=3, recover=yes];

    //Deny everything else
    deny all;
  };
}
```

Authorisations specified in subtasks override authorisations in the parent task for the period of time which that sub-task is executing.

`tudo` tasks are similar to the process-specific file protection provided in the TRON system[3]. In TRON, system call wrappers are used to provide a protection domain in which normal processes can run with restricted privileges. Like TRON, `tudo` allows rights to be specified on a per process basis (as identified by the program and arguments used to create that process), but also provides the capability to roll back actions.

`role` A list of users and groups along with their access privileges. e.g. The role "SpaceClearer" which comprises the engineering and admin group and are allowed to delete "junk" files to clean up space can be specified as:

```
type junk_files {
  class: file;
  path: ("/home/*/core",
         "/home/*/*~",
         "/home/*/.netscape/cache",
         "/home/*/.netscape/cache/*");
};

role SpaceClearer {
  members: (engineering, admin);
  rules: {
    allow search "/home/*";
    allow rm junk_files;
    allow exec ("/bin/find",
               "/bin/xargs",
               "/bin/rm");

    //Everything else is disallowed
```

```
        deny all;
        }
    };
```

5.5.2 Role-Based Access control

RBAC is an access control paradigm in which access decisions are based on the functions a user is allowed to perform within an organisation, rather than "data ownership" [7]. RBAC is supported in TUPL by allowing the administrator to define roles using the `role` construct. A role can include users, groups or other roles as members. Each role specifies the access control rules associated with it. In addition, other access control rules can be defined to apply to roles. TUPL does not however, support conditional membership rules, and roles are not dynamically invoked. A user must specify which role they are adopting when invoking a `tudo` session.

5.5.3 Domain and Type Enforcement

Domain and Type Enforcement (DTE) allows objects and subjects in the system to be typed, and authorisations to depend on these types [1]. Subjects perform actions in *domains*, and the DTE system provides rules for how transitions to new domains can occur. DTE is supported in TUPL using the `type`, and `task` constructs. In TUPL, the concept of a `task` roughly equates to a DTE domain, and the `type` constructs equate to a DTE type. By allowing tasks and sub-tasks to be dynamically invoked, TUPL supports a limited way of constraining transitions between domains. However, this is not as flexible as that supported in other DTE mechanisms (e.g. [1]).

6 Evaluation

In order to determine the usefulness of the `tudo` utility, it must be evaluated to determine how well it meets the design criteria in section 4. In particular, it must enforce well-formed and partially-formed transactions in accordance with the models presented.

6.1 Enforcement of well-formed transactions

Providing well-formed transactions requires the enforcement of the four rules described by Clark and Wilson.

- E1 – Only certified TPs can operate on CDIs

  In order to perform an action it must be specifically authorised in the policy file. Because only certified actions are assumed to be authorised, this meets the requirement for rule E1.

- E2 – Users must be authorised

  This rule is also enforced by the policy mechanism.

- E3 – Users must be authenticated

  This rule is enforced by requiring the user to enter their password in order to perform an action.

- E4 – Only administrator can specify TP authorisations (i.e. MAC)

  `tudo` provides a specially administrator privilege which is required to edit the policy file to ensure this rule is enforced.

6.2 Enforcement of partially-formed transactions

Enforcement of partially-formed transactions requires that the system observe the five enforcement rules described in [16].

- E1 – The system must ensure that only TPs which have been recoverable can be run

  This rule is enforced for all actions where the recover=yes option is set. If recover=yes is set for an option which is not recoverable, that action will fail (this can happen for example when using the rule `allow all [recover=yes]`.

- E2 – Users must be authorised

  This is enforced by the policy mechanism.

- E3 – Users must be authenticated

  This is enforced by requiring a user to enter their password in order to perform an action.

- E4 – Transactions must be logged

  This is enforced by the logging mechanism in tudo.

- E4 – Only administrator can specify TP authorisations (i.e. MAC)

  tudo provides a specially administrator privilege which is required to edit the policy file to ensure this rule is enforced.

6.3 Security

One of the most important lessons learned in implementing tudo is that the *principal of least privilege* is sagely advice. Implementing an optimistic system which relaxes this requirement raises many challenges. These issues are discussed below.

6.3.1 Programming practices

tudo has been implemented using good programming practices to ensure that running it as a setuid application does not open up any security vulnerabilities. In particular tudo ensures that all system call returns are checked, and that only "safe" version of library functions for reading and manipulating user input are used. In addition, tudo sanitises environment variables by setting them to safe values.

6.3.2 Self protection measures

It should be noted that the enforcement mechanism that tudo uses is rather fragile. If a user can find a way to kill the tracing process without killing the programs it is tracing, then they will be able to perform actions without being subject to the access control mechanism. To help prevent this, tudo sets the *kill-on-last-close* flag for each process it traces, so that if tudo exits for some reason, all traced processes will be automatically killed with a SIGKILL signal. In addition, tudo disassociates itself from its parent process group to ensure that killing parent processes does not kill tudo. Finally, tudo contains an implicit rule which prevents any traced application from sending a SIGKILL signal to any tudo process, or from opening any file in a tudo processes /proc filespace for write access.

6.3.3 Race conditions

As discussed in section 5.2, a potential race condition exists between checking the authorisation and allowing the system call to proceed. tudo protects against this by checking the status of some objects after the system call has completed and killing the process if they do not match expectations.

6.3.4 Handling a full secure logical partition

tudo has a hardcoded high watermark on the capacity of the logical partition, and will abort a session and roll back the intermediate states once this high watermark is exceeded. This stops a user who might have write permission to the physical volume on which the secure logical partition is stored from filling up the disk in order to prevent tudo from logging information. Once the high watermark is exceeded, tudo requires that log state be cleaned up to below a low watermark before any more activity is allowed (with one exception - to allow the administrator to specify tudo tasks which cycle the logs etc, tudo will allow those actions/task with the option `cleanup=yes` set to operate while the high watermark has exceeded).

6.3.5 Handling symbolic and hard links

Handling of filesystem links is somewhat complex in tudo. Because file authorisations are done on the basis of filename patterns, it could be possible for a user to create a symbolic link to a restricted file in an unrestricted filespace. For this reason, tudo always applies access restrictions on the real filename for symbolic links.

A more complex situation occurs for hard links where two different files share the same inode on the same filesystem. It is difficult for tudo to determine the names of all files linked to a given inode without performing a scan of the raw disk device. For this reason, tudo allows accesses on files with multiple hard links to occur, but logs a warning message to indicate that the access is potentially suspicious.

6.3.6 Constraining non-syscall access

The Unix operating system provides other ways to manipulate the system other than system calls. Pseudo files such as `/dev/kmem`, `/dev/mem`, `/proc` and the the raw disk device files can provide ways for users to bypass the security mechanisms provided by `tudo`. Again, because such accesses can be legitimate, the administrator must explicitly deny users access to such files through a rule such as:

```
deny all ("/dev/*", "/proc/*");
```

6.4 Limitations

There are some limitations with the `tudo` prototype. Using the system tracing facility of the `/proc` filesystem means that a file descriptor must be open for every process being traced. If a large number of processes are being executed (for example in a shell script), `tudo` may be unable to complete the task.

In addition, determining the privileges necessary to complete a particular task can be time-consuming and difficult. An administrator can simplify the process by creating a task with a command entry point and authorising all actions for that command, but this creates problems if the command is not entirely trusted. Often an administrator will need to use a system call tracing facility such as `strace` or `truss` to determine what accesses an application needs.

A further limitation is that `tudo` does not provide the ability for administrators to specify how actions should be rolled back. Currently, recovery is implemented by tracking changes to modified files and reverse applying the changes. However, there may be some higher level semantics which require such actions to be compensated in a different way.

6.4.1 Lack of transactional semantics for `tudo` actions

Perhaps the biggest limitation of `tudo` is that it does not provide full transactional semantics (i.e. ACID properties) for actions, and as such it is possible for programs to either see or modify intermediate states. This arises for two reasons:

1. `tudo` itself does not perform any isolation or concurrency control of `tudo` sessions, and as such two `tudo` sessions running at the same time may create conflicts.
2. Users may perform other operations without using `tudo`, but which either read or write objects accessed in `tudo` sessions. As these actions are neither logged nor recoverable, a user can use their `tudo` privileges to engage in seemingly legitimate behaviour, but can run a simultaneous session which exploits insecure intermediate states.

This problem is indicative of the lack of transactional semantics in Unix in general. Because the security of both well-formed and partially-formed transactions are based on the assumption of atomic and isolated actions, this short-coming is problematic for `tudo`, and can lead to situations in which privileged access can bypass the reference monitor.

Of the two reasons given above, the second case is perhaps the most difficult for an optimistic system. Recall that for partially-formed transactions, consistency is only guaranteed by the existence of a compensating mechanism which can be used to recover the system to a valid state. If we compare this situation to a traditional database system, we see that the commit phase is analogous to a system administrator reading a `tudo` log, and judging whether an action should be accepted or rolled back. However, because `tudo` exposes the intermediate state, it may be the case that either rolling back will not undo all the damage that has been done, or that the administrator will be unaware that a conflict exists.

A simple example which illustrates this problem is when a user creates a setuid executable, or downgrades the file permissions on an important file (`/etc/passwd` for example) using `tudo`. While `tudo` can roll back these actions, the system administrator can never be sure what the user did while these backdoor holes were in place. The user might legitimately argue that the permission change was an accident and that there was no malicious intent.

The solutions to the first problem are relatively straight forward applications of traditional concurrency control. `tudo` could implement its own optimistic timestamping mechanism and use its existing rollback mechanism to cope with conflicts. However, while the solutions to the second problem are simple in theory, they are pragmatically less desirable. One approach would be to require that all users run tudo as their default shell so that all actions are logged and recoverable. This doesn't close all the holes (e.g. setuid programs which are able to leverage root privilege to work around tudo's controls as described above are still a problem, and a user might still be able to rewrite their shell in the password file), nevertheless, it does improve the situation. However, users may be unwilling to pay the inevitable performance penalty which comes with tudo for resource/CPU intensive applications. Another approach which is maybe more palatable from this aspect, but at the cost of reducing the flexibility of the mechanism, is to write rules into tudo to ensure that no object can have its privileges downgraded and no principal can have their privileges upgraded *except* during a given tudo session. This would mean any changes to a file's permissions for example, would have to be undone before a tudo session exited, otherwise the entire session would be aborted and the actions would be rolled back.

6.5 Other uses

This paper has described the use of tudo to enforce well-formed and partially-formed transactions. However, tudo can also be used for other purposes. tudo can also provide a sandbox for untrusted applications such as that provided by the Janus environment[11]. In addition, tudo could also be used to run setuid shell scripts, similar to super[20].

7 Related Work

In [5], Clark and Wilson compare their integrity model to others based on the Bell and LaPadula secrecy model [2]. Because partially-formed transactions are closely related to well-formed transactions, much of this discussion also applies to them.

Foley[8] examines various integrity models (including Clark-Wilson) and argues that these models are limited in the sense that they only consider integrity in an operational/implementation sense. Foley presents a formal model in which integrity is considered as just one attribute of a *dependable* system. He shows that dependability can be seen as a form of refinement, in which the system can implement top-level requirements in the presence of failures, and demonstrates how this model can be used to describe concepts such as separation of duties, and assured pipelines as implementation techniques for achieving integrity. Because partially-formed transactions are simply another such implementation technique, they could be easily modelled using Foley's scheme.

As discussed, Goldberg et al [11] describe an environment similar to tudo which is used to provide a *sandbox* for untrusted helper applications. While their design approach is similar to that taken by tudo, Janus differs in the fact that it does not provide the logging and recovery features which are central to tudo's architecture.

Numerous access control systems have been implemented to augment Unix's simple system, some of which are discussed in [11]. Notable works include: [3, 1, 6]. These systems are primarily aimed at providing richer access control semantics for Unix, and do not provide the recovery features necessary to implement partially-formed transactions. These systems have also been implemented using kernel modifications, or as system call wrappers which would seem to be a less flexible (although possibly more performant) mechanism than is used for tudo.

Recovery has been well studied in the area of concurrency control and transaction processing. Ramamritham and Panos [17] give an excellent overview of the state of the art in this area. Partially-formed transactions are similar to *sagas*[10], as they are long running, and consist of a number of independent component transactions. Like partially-formed transactions, compensating actions are used in sagas to maintain consistency. Partially-formed transaction also have something in common with *transactional workflows*, where a desire to relax the ACID properties of a transaction mean that system failures must be dealt with using compensating transactions [13]. It would be useful to investigate the techniques used in these more exotic transactional systems to see how they might improve some of the limitations with the tudo prototype.

8 Future Work

This paper has focussed on the issues of how to provide both pessimistic and optimistic access control using well-formed and partially-formed transactions in a Unix environment. However it would be useful to extend this work by looking at how different authorisation models such as n of m threshold schemes might help to minimise the risks of misuse by requiring more than one person to request a questionable action before it is authorised.

Implementing rollback for network and interprocess communications is another interesting problem, as compensating transactions in this case are not clear cut. Some investigation of how to recover from such actions would help to extend the concept of partially-formed transactions to these domains.

Finally, performance issues in the `tudo` prototype should be addressed. These are particularly problematic in the case of modifying very large binary files, as `tudo` currently needs to take a full copy of these files each time they are opened for write. It would also be useful to see how `tudo` could work with a log-type filesystem where the ability to roll back changes is supported by the filesystem, rather than as an add on using `tudo`.

9 Summary

This paper has motivated the need for a mechanism to allow users to not only perform legitimate actions, but to allow for cases in which certain questionable actions should be authorised. It describes an optimistic access control mechanism which seeks to maintain the integrity of transactions by providing a mechanism which allows actions to be rolled back to a valid state. The concept of partially-formed transactions has been formally described, and the paper shows how such transactions can be used to enforce the security properties of integrity, access control, audibility and accountability for an optimistic system.

A proof-of-concept prototype has been designed and implemented for the Solaris operating systems. It demonstrates how a reference monitor can be constructed using the system call tracing facilities of the `/proc` filesystem which enforces both well-formed and partially-formed transactions. Security features and limitations of the prototype are discussed, and future directions are discussed for this research.

10 Acknowledgements

The work reported in this paper has been funded in part by the Co-operative Research Centre Program through the Department of the Prime Minister and Cabinet of Australia.

11 Availability

The `tudo` prototype is available from

http://security.dstc.edu.au/projects/tudo/

References

[1] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical domain and type enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, 1995.

[2] D. E. Bell and L.J. LaPadula. Secure computer systems. Technical Report ESD-TR-73-278 (Vol I-III) (also Mitre TR-2547), Mitre Corporation, Bedford, MA, April 1974.

[3] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 USENIX Winter Technical Conference*, pages 165–175, 1995.

[4] M. Bishop. Unix security: Security in programming. *SANS'96*. Washington DC, May 1996. URL: http://seclab.cs.ucdavis.edu/-bishop/secprog.html.

[5] David D. Clark and David R. Wilson. A comparison of commercial and military security policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[6] G. Fernandez and L. Allen. Extending the Unix protection model with access control lists. In *Proceedings of the Summer 1998 USENIX Conference*, pages 119–132, 1988.

[7] David Ferraiolo and Richard Kuhn. Role-based access control. In *Proceedings of the 15th National Computer Security Conference*, 1992.

[8] Simon N. Foley. Evaluating system integrity. In *New Security Paradigms Workshop*. ACM press, 1998.

[9] Peter Galvin. The Unix secure programming faq - tips on security design principles, programming methods and testing. *Sunworld*, August 1998. URL: http://www.sunworld.com/swol-08-1998/swol-08-security.html.

[10] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 249–259, New York, NY, 1987. ACM press.

[11] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications – confining the wily hacker. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, California, July 1996.

[12] John D. Howard. *An Analysis Of Security Incidents On The Internet:1989 - 1995*. PhD thesis, Carnegie Mellon University, April 1997. URL: http://www.cert.org/research/JHThesis/Start-.html.

[13] Dean Kuo, Michael Lawley, Chengfei Liu, and Maria Orlowska. A model for transactional workflows. *Australian Computer Science Communications*, 18(2):139–146, 1996.

[14] Dennis Longley, Michael Shain, and William Caelli. *Information Security – Dictionary of concepts, standards and terms*. Macmillan, 1992.

[15] Peter G. Neumann. *Computer related risks*. Addison-Wesley, 1995.

[16] Dean Povey. Optimistic security: A new access control paradigm. In *Proceedings of the 1999 New Security Paradigms Workshop*, September 1999. to appear.

[17] Krithi Ramamritham and Panos K. Chrysanthis. *Executive Briefing: Advances in Concurrency Control and Transaction Processing*. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[18] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[19] sudo. URL: ftp://ftp.courtesan.com/pub/sudo.

[20] super. URL: ftp://ftp.ucolick.org/pub/users/will/.

[21] *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense Computer Security Center, Fort Meade, MD, August 1983.

Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications*

R. Sekar P. Uppuluri
State University of New York at Stony Brook, NY 11794.
{sekar,prem}@cs.sunysb.edu

Abstract

To build *survivable information systems* (i.e., systems that continue to provide their services in spite of coordinated attacks), it is necessary to detect and isolate intrusions *before* they impact system performance or functionality. Previous research in this area has focussed primarily on detecting intrusions after the fact, rather than preventing them in the first place. We have developed a new approach based on specifying intended program behaviors using patterns over sequences of system calls. The patterns can also capture conditions on the values of system-call arguments. At runtime, we intercept the system calls made by processes, compare them against specifications, and disallow (or otherwise modify) those calls that deviate from specifications. Since our approach is capable of modifying a system call before it is delivered to the operating system kernel, it is capable of reacting before any damage-causing system call is executed by a process under attack. We present our specification language and illustrate its use by developing a specification for the ftp server. Observe that in our approach, every system call is intercepted and subject to potentially expensive operations for matching against many patterns that specify normal/abnormal behavior. Thus, minimizing the overheads incurred for pattern-matching is critical for the viability of our approach. We solve this problem by developing a new, low-overhead algorithm for matching runtime behaviors against specifications. A salient feature of our algorithm is that its runtime is almost independent of the number of patterns. In most cases, it uses a constant amount of time per system call intercepted, and uses a constant amount of storage, both independent of either the size or number of patterns. These benefits make our algorithm useful for many other intrusion detection methods that employ pattern-matching. We describe our algorithm, and evaluate its performance through experiments.

*This research is supported in part by Defense Advanced Research Agency's Information Technology Office (DARPA-ITO) under the Information System Survivability program, under contract number F30602-97-C-0244.

1 Introduction

Our increasing reliance on networked information systems to support critical infrastructures (e.g, telecommunication, commerce and banking, power distribution, and transportation) has prompted interest in making the information systems *survivable*, so that they continue to perform their primary functions even in the face of coordinated attacks. In order to build survivable systems, it is necessary to detect and isolate attacks *before* they impact system performance or functionality. Thus a number of recent research efforts have focussed on the problem of *intrusion prevention* [GWTB96, SBS99, CPMWBBGWZ98, MLO97, FBF99].

Many known approaches for intrusion prevention (including the one described in this paper) are based on the following observation about attacks: regardless of the nature of an attack, damage can ultimately be caused only via system calls made by processes running on the attacked host. It is hence possible to prevent damage due to attacks if we can monitor every system call made by every process, and prevent damage-causing calls from being executed. Actions to contain or respond to the attack could be launched at this point, e.g., terminating the process. Some of the central problems in the context of these approaches are:

- efficient interception of system calls
- pattern languages to characterize normal/abnormal system call sequences for any given program/process
- efficient matching of actual system calls made by a process against such patterns

Several techniques for system call interception have already been proposed in [GWTB96, MLO97, GPRA98, FBF99]. This paper addresses the remaining two issues, focusing particularly on an expressive and easy-to-use specification language, and pattern-matching algorithms that are fast enough to be invoked on every system call.

1.1 Summary of Results

- In Section 2 we present a new and expressive language for capturing patterns of normal or abnormal behaviors of processes in terms of sequences of system calls and their arguments. As compared to the language described in [SCS98, SBS99], this paper focuses on a core language that we call regular expressions for events (REE). REEs extend regular expressions to model system calls that are characterized by a name as well as argument values. Response actions can be associated with patterns, and these will be launched automatically when our system observes a match for the pattern.
- In Section 3 we illustrate our language with several simple examples. We then detail the process of developing a complete specification for the `ftpd` server. This example, together with similar specifications for `httpd` and `telnetd` forms the basis of our experimental results in later sections.
- REE have much higher expressive power than regular expressions. The presence of variables makes them comparable to attribute grammars in terms of expressive power. The examples in Section 3 illustrate that in practice, REE language provides the capabilities needed to describe program behavior as needed for intrusion detection or prevention.
- In Section 4 we present our runtime model for fast pattern-matching, based on extended finite-state machines (EFSA). Just as REE's extend the power of regular expressions to permit variables, EFSA extend traditional finite-state automata to be able to assign or examine values of a finite set of variables. As in the case of regular expressions, every REE can be matched by a non-deterministic EFSA (NEFA) whose size is linear in the size of REE. But simulation of a NEFA at runtime can be very inefficient. We then examine the properties of deterministic EFSA (DEFA) which can be simulated efficiently. However, we show that the size of deterministic EFSA can be far too large (super-exponential in the size of NEFA) for the approach of using DEFA to be viable.
- Ideally, we would like an approach that achieves a balance between the space explosion implied by DEFA with the runtime inefficiency imposed by NEFA. We propose an algorithm to accomplish this in Section 5. Given an REE of size N, our algorithm produces an EFSA with a worst-case size that is $O(2^N)$. We show that this EFSA is deterministic for a subclass of REE's called REE(0) patterns.
- We present an implementation of our techniques in Section 6 and report its performance. In practice, our algorithm uses much less than exponential amount of space. This means that we can perform intrusion prevention/detection in essentially constant time per system call for most patterns. Our current implementation introduces about 1 to 2% overhead due to the pattern-matching operations.

The applicability of the techniques developed in this paper extends well beyond our system. For instance, many intrusion detection techniques (e.g., [Kumar95, PK92, Ko96]) are formulated using signature patterns that characterize attacks. In contrast with our approach, the performance of matching algorithms developed in these approaches worsens linearly with the number of patterns. By using the techniques developed in this paper, the performance of these approaches can be improved significantly.

2 Approach Overview

We model security-related behaviors in terms of sequences of (security-related) events. In general, events may be internal to a single process, e.g., a function call made by the process; or they may be externally observable, e.g., a message delivered to a machine, or a system call made by a process. We are particularly interested in the system call events, since it is possible to *enforce* secure behaviors if we can intercept and modify system calls. Our specifications characterize normal and/or abnormal behavior of programs as patterns over system call sequences. These specifications are compiled into optimized programs for efficient detection of deviations from the specified behavior. When discrepancies are detected at runtime, defensive actions are initiated to contain or isolate the damage.

Our intrusion detection/prevention system consists of an offline and a runtime component. The offline system generates detection engines from specifications, while the runtime system provides the execution environment for these engines. The input to the offline component is a specification S_P for each program P to be defended[1]. The offline component

[1]This specification may be developed based on known information about P, obtainable from such sources as manual pages, security advisories etc.

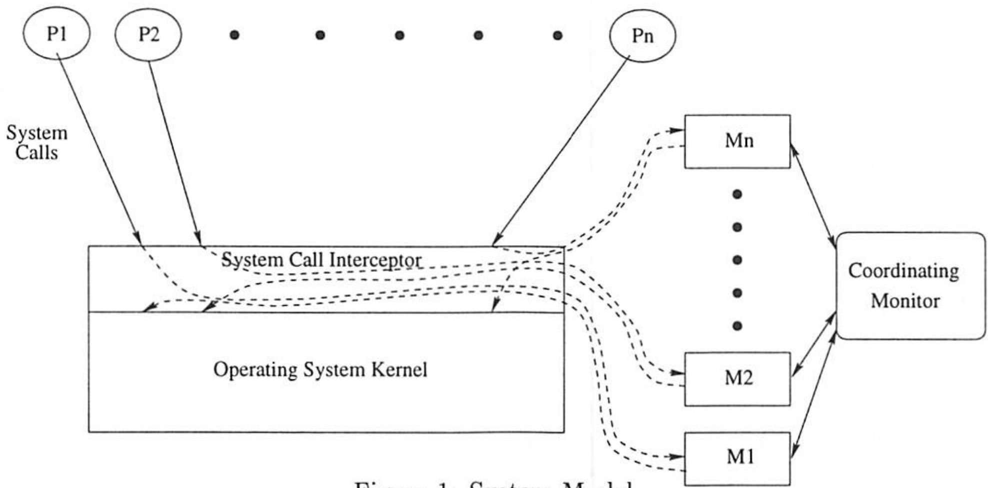


Figure 1: System Model

translates S_P into a C++ class definition C_P. This C++ class simulates an extended finite state machine for matching the patterns in S_P and initiating appropriate responses to intrusion attempts. This code is compiled and then linked with a runtime infrastructure to produce a detection engine.

Figure 1 shows the runtime operation of our system. When P executes as process P_j, it is monitored using the detection engine M_j, which incorporates an instance of C_P. System calls made by P_j are intercepted by the system call interceptor just before, and just after the system call's kernel level functionality is executed, and the system call information is passed to M_j. If M_j matches a pattern, it invokes the action associated in S_P with that pattern. This action would typically utilize the support functions provided by the runtime system to modify the just-initiated system call execution so as to prevent it from causing damage.

The *fork* system call is given special treatment. When process P_j executes a fork system call, it results in two processes P_j and P'_j. At this point, the detection engine monitoring P_j also clones itself, with one instance monitoring P_j and another monitoring P'_j. If one of these processes, say P'_j, executes another program using the *execve* system call, we may switch to monitoring the new program with respect to a new specification as described later.

Frequently, security specifications for a process are completely independent of the actions of other processes. However, some specifications concern interactions among multiple processes. To monitor the behavior of concurrent processes, we use a coordinating monitor that communicates with the monitors for different processes to identify deviant behaviors. We express specification of process interaction using a language construct called an atomic sequence, described later. The coordinating monitor is used to detect violations of atomic sequences. While we describe the atomic sequence construct and discuss a possible implementation to support it, we have not yet implemented it. As such, we do not provide experimental results regarding this construct.

2.1 Specification Language

We provide an overview of the principal components of our specification language here. The main components of a specification include variable declarations and rules. Variables declared in this manner are global, as opposed to *local variables* whose scope is limited to a single rule. Such global variables are referred to as *state variables*, while local variables are called *temporary variables*. State variables are restricted to be of primitive types, i.e., long, integer, boolean, char, and unsigned versions of these. A rule is of the form $pat \rightarrow actions$. Here, *pat* denotes a pattern on sequences of system calls. When a process is monitored using this specification and the process makes a sequence of system calls that matches *pat*, the responsive steps contained in *actions* is initiated.

2.1.1 Event Patterns

Event patterns are built from events using sequencing operators. Events are of the form $EventName(Arg_1, ..., Arg_n)$. For each system call, we identify two events: the entry event which corresponds to the invocation of the system call, and the exit event which corresponds to the return from the system call. The entry event uses the same name as that of the system call, while the exit event is obtained by appending `_exit` to the system call name. $Arg_1, ..., Arg_n$ denote the system call arguments. An *event history* is a sequence of events.

We define a special event *begin* which precedes any system call made by any process, and the event pattern *any* that stands for any event.

Sequencing operators are similar to those used in regular expressions, but operate on events with arguments. We refer to our pattern language as regular expressions over events (REE) to indicate this relationship. *Elementary patterns* in our language are of the form $e(x_1, ..., x_n)|cond$, where *cond* is a boolean-valued expression on the event arguments $x_1, ..., x_n$, any temporary variables that may appear earlier in a pattern, and state variables. The condition component can make use of standard arithmetic, comparison and logical operations and several support functions. The support functions allowed in a pattern correspond to "read" operations that do not modify the state of the monitored process. An example of such a function is $realpath()$ which translates a file name into a canonical form that does not contain ".", "..", or symbolic links.

The meaning of event patterns and the sequencing operators is best explained by the following definition of what it means for an event history to match a pattern:

- *event occurrence:* $e(x_1, ..., x_n)|cond$ is satisfied by the event history $e(v_1, ..., v_n)$ if *cond* evaluates to *true* when variables $x_1, ..., x_n$ are replaced by the values $v_1, ..., v_n$.
- *event nonoccurrence:* $!e(x_1, ..., x_n)|cond$ is matched by H if it does not match $e(x_1, ..., x_n)|cond$.
- *sequencing:* $pat_1; pat_2$ is satisfied by an event history H of the form H_1H_2 provided H_1 satisfies pat_1 and H_2 satisfies pat_2.
- *alternation:* $pat_1 || pat_2$ is satisfied by an event history H if either pat_1 or pat_2 is satisfied by H.
- *repetition:* $pat*$ is satisfied by an event history $H_1H_2 \cdots H_n$ iff H_i satisfies pat, $\forall\ 1 \leq i \leq n$.
- *realtime constraints:* *pat* **within** t is satisfied by an event history H_1 if H_1 satisfies *pat* and the time interval between the first and last events in H_1 is less than or equal to t.
- *atomicity:* **nonatomic** d **in** *pat* denotes that accesses to data d be atomic in *pat*, i.e., without any intervening operations by other processes that could modify this data.

When a variable occurs multiple times within a pattern, an event history will satisfy the pattern only if the history instantiates all occurrences of the variable with the same value. For instance, the pattern $e_1(x); e_2(x)$ will not be satisfied by the event history $e_1(a)e_2(b)$, but will be satisfied by $e_1(a)e_2(a)$.

We relax the definition of the sequencing operators to minimize the need to include both entry and exit events in a pattern. For instance, when two system call entry events occur in sequence, we implicitly insert an exit event in the middle, e.g., $(open|C_1); (close|C_2)$ is treated as equivalent to $(open|C_1); open\_exit; (close|C_2)$. If the first event is an entry event while the second is an exit event for a different system call, then an exit event for the first call and the entry event for the second call are implicitly added, e.g., $(open|C_1); (close\_exit|C_2)$ is treated as equivalent to $(open|C_1); open\_exit; close; (close\_exit|C_2)$.

The value of a temporary variable should be defined before its first use via a *binding* that takes the form $tvar = expr$. Note that binding differs from assignment in that there can be at most one binding to any temporary variable, as subsequent conditions of the form $tvar = expr$ are treated as comparisons.

2.1.2 Response Actions

The response action associated with a rule $p \rightarrow a$ is launched if *a suffix of the event history matches p*. The action component consists of a sequence of statements, each of which can either be an assignment to a state variable or invocation of a support function provided by the runtime system[2].

A typical response is to prevent a system call from executing and/or return a fake return value. This is accomplished using a support function *fail*, which takes an argument that corresponds to the value to which the variable *errno* should be set to. Sometimes, it is necessary (or just convenient) to switch to a new specification using the support function $switch(f)$, where f is the name of the new specification.

If multiple patterns match at the same time, all of the associated actions are launched. This leads to a problem when some of these actions conflict with each other. It is possible to address such interactions by (a) defining a notion of *conflict* among operations contained in the reaction components of

[2]Knowledge about these support functions are not integrated into the specification language, but are declared in header files that can be included in the specification. Due to space constraints, we do not treat these declarations in detail.

rules, whether they be assignments to variables or invocation of support functions provided by the runtime system, and (b) by stipulating that there must not exist two patterns with conflicting operations such that for some sequence of system calls, they can match at the same point. Potential conflicts can be identified by the automaton construction algorithms developed in this paper — if there is any state in the automaton that corresponds to a final state for two such patterns, then there is a potential conflict. However, we have not implemented this approach yet, instead relying on the specification writer to deal with such conflicts.

3 Example Specifications

We begin this section with a few simple examples and then proceed to give a complete specification for the FTP daemon.

3.1 Simple Examples

To restrict a program from making a set of system calls, we can create a simple pattern that matches any of these disallowed system calls and then invoke an action that causes these calls to fail. For instance, we may wish to prevent a server program such as `fingerd` from executing any program, modify file permissions, create files or directories or initiate network connections. We use the shorthand notation of omitting some of the trailing (and sometimes, all of the) variables of a system call when we are not interested in their values.

$$\begin{array}{l} execve || connect || chmod || chown || \\ \quad creat || truncate || sendto || mkdir \\ \rightarrow fail(EINVAL) \end{array}$$

We may also wish to restrict the files accessed for reading or writing. For a program such as `fingerd`, we may use the following rule to prevent the program from writing any file, and reading any files other than those mentioned in `admFiles` defined below.

$$\begin{array}{l} admFiles = \{\texttt{"/etc/utmp",} \\ \quad \texttt{"/etc/passwd", datadir/*}\} \\ open(f, mode) | (realpath(f) \notin admFiles \\ \quad || \ (mode \neq \texttt{O\_RDONLY}) \\ \rightarrow fail(EPERM); \end{array}$$

To illustrate the use of sequencing operators, consider the following pattern that asserts that a program never opens and closes a file without reading or writing into it. Before defining the pattern, we define abstract events that denote the occurrence of one of many events. Occurrence of an abstract event in a pattern is replaced by its definition, after substitution of parameter names, and renaming of variables that occur only on the right-hand side of the abstract event definition so that the names are unique.

$$\begin{array}{l} openExit(fd) ::= \\ \quad open\_exit(f, fl, m, fd) || creat\_exit(f, m, fd) \\ rwOp(fd) ::= read(fd) || readdir(fd) || write(fd) \\ openExit(fd); (!rwOp(fd))*; close(fd) \rightarrow \cdots \end{array}$$

Although regular expressions are not expressive enough to capture balanced parenthesis, the presence of variables in REE enables us capture the close system call matching an open. This issue of expressive power is discussed again later.

The example below illustrates the use of atomic sequence patterns. A popular attack uses race conditions in setuid programs as follows. Since the setuid process runs with effective user *root*, any open operation by the process will always succeed. If the process is running on behalf of a user, and wishes to open a file with the permissions of this user (i.e., with privileges corresponding to the real userid), it may do so by first using the *access* system call which determines if the real user has access to a file, and if so, it goes ahead and opens the file. The attacker exploits the time window between the access and open system calls as follows. The attacker uses a symbolic link as the name of the file in question, and changes the target of the link between the access and open system calls. To prevent this attack, we ensure that the object referred by the *access* and *open* system calls is accessed atomically:

$$\begin{array}{l} nonatomic(f.target) \text{ in} \\ \quad (access(f); (!open(f))*; open(f)) \\ \rightarrow fail(EACCES); \end{array}$$

3.2 Case Study: Specification for `ftpd`

Our starting point in developing a specification of `ftpd` is the documentation provided in its manual pages. Specifically, we identified the following properties for wu-ftpd by examining its manual page and based on our knowledge of UNIX. These properties are captured in our specification language in Figure 2. Although it is possible to turn the English descriptions directly into specifications in our language, it is usually necessary to cross-check (or "debug") the specifications by monitoring ftpd under typical conditions. We have therefore used a hybrid approach, where we first manually inspected system call traces produced by ftpd, and used it to

```
    /* Define useful constants.  */
1.  ftpAdmFilePrefixes ::= {/etc/,/lib/,/usr/lib/,/dev/null/,/var/run/ftp}
2.  ftpInvalidUsers ::= {0,BINUID,SYSUID,MAILUID}
3.  ftpInvalidPutDirs ::= {/, /bin/, /sbin/, /usr/, /etc/}
    /* Define useful abstract events. We assume that certain abstract events such as privileged (which denotes */
    /* certain privileged system calls that are not used by most programs) and wrOpen (which denotes */
    /* any file open operation that can create or modify the file).  */
4.  ftpInitBadCall ::= (wrOpen(f)|(f != /dev/null) && !isExtension(/var/run/ftp,f))||permChange()||
       rename()||link()||delete()||mkdir||rmdir||admin()||execve||clone||
       bind||listen||connect||accept||recvfrom||recvmsg||sendto||sendmsg
5.  ftpAccessBadCall ::= admin||accept||recvfrom||recvmsg||sendto||sendmsg||clone
6.  ftpPrivCalls ::= close||uidgidops||socket||setsockopt||(bind(s,sa)|port(sa)=FTPDATAPORT)
7.  ftpValidExecs ::= {/bin/ls,/bin/tar,/bin/gzip}
8.  ftpAccessedSvcs ::= {NAMESERVER}
    /* Use a state variable to remember the uid of user logging in and client host name */
9.  int loggedUser := NOBODYUID
10. int clientIP := 0
11. begin();(!setreuid)*;setreuid(r,e) → loggedUser := e
12. begin();(!getpeername)*;getpeername_exit(fd,sa,l) → clientIP := getIPAddress(sa)
    /* Host authentication phase must precede user authentication.  */
13. begin();(!getpeername)*;open(/etc/passwd) → term()
    /* User authentication must precede before userid changed to that of the user.  */
14. begin();(!open(/etc/passwd))*;setreuid() → term()
    /* Access limited to admin-related files before user login is completed.  */
15. begin();(!setreuid())*;open(f)|(!isExtension(ftpAdmFilePrefixes, f)) → term()
    /* Access limited to certain system calls before user login.  */
16. begin();(!setreuid())*;ftpInitBadCall() → term()
    /* Certain system calls are not permitted after user login is completed.  */
17. setreuid();any()*;ftpAccessBadCall() → term()
    /* Anonymous user login: must do chroot before setreuid.  */
18. begin();(!(setreuid||chroot(FTPHOME)))*;setreuid(r,FTPUSERID) → term()
    /* Userid must be set to that of the logged in user before exec.  */
19. begin();(!setuid(loggedUser))*;execve → term()
    /* Resetting userid to 0 is permitted only for executing a small subset of system calls.  */
20. setreuid(r,0);ftpPrivCalls*;
       !(setreuid(r1,loggedUser)||setuid(loggedUser)||ftpPrivCalls) → term()
    /* Any file opened with superuser privilege is either explicitly closed before an exec, or has close-on-exec flag set.  */
21. (open_exit(f, fl, md, fd)|geteuid()=0);(!close(fd))*; (execve|!closeOnExec(fd)) → term()
    /* Site-specific: ensure that ftp cannot be used to write files into certain directories.  */
22. wrOpen(f)|(f ∈ ftpInvalidPutDirs) → term()
    /* Site-specific: ensure certain users cannot login using ftpd.  */
23. begin();(!setreuid)*;setreuid(r,e)|(e ∈ ftpInvalidUsers) → term()
    /* Site-specific: ensure ftp cannot execute arbitrary programs.  */
24. execve(f)|(f ∉ ftpValidExecs) → term()
    /* Site-specific: ftp cannot connect to arbitrary hosts or services.  */
25. connect(s, sa)|((getIPAddress(sa) != clientIP)&&(getPort(sa) ∉ ftpAccessedSvcs)) → term()
```

Figure 2: A specification for `ftpd`.

further narrow down the actions/behaviors that the ftpd server may exhibit. In most cases, we have not attempted a sophisticated response, instead opting for a simple action such as terminating the process using a support function named *term*().

- ftpd attempts to authenticate a client host before proceeding to user authentication phase. Precisely identifying the sequence of system calls that correspond to client authentication is hard, as it involves a large number of steps that may vary from installation to installation. As such, we treat *getpeername* as a marker that indicates host authentication related processing. Similarly, we treat opening of */etc/passwd* as a marker for user authentication related processing. Rule 13 captures this English description by stipulating that an open of the password file should never happen before invocation of *getpeername*.

- users need to be first logged in before most files can be accessed. Rule 14 uses *setreuid* as an indicator for completion of user login process.
- after user authentication is completed, ftpd sets the userid to that of the user that just logged in. We remember this userid for later use (rule 11).
- prior to user authentication, only files beginning with names identified in the set `ftpAdmFilePrefixes` can be accessed (rule 15).
- certain system calls are never used before user login, and certain others are never used after login process (rules 16, 17). Let `ftpInitBadCall` denote a pattern that matches system calls not used prior to user login. Similarly, let `ftpAccessBadCall` match system calls that are not used after login[3].
- for anonymous login, the userid `FTPUSERID` is used; moreover, the `chroot` system call is used to restrict access only to the subtree of the filesystem rooted at `~ftp` (rule 18).
- ftpd resets its effective userid to root in order to bind certain sockets to ports numbered below 1024. The userid is reverted back to that of the logged in user immediately afterwards (rule 20).
- to eliminate possible security loopholes, `ftpd` must execute a setuid system call to change its real, effective and saved userid permanently to that of the logged in user before executing any other program; otherwise, the executed process may be able to revert its effective userid back to that of superuser (rule 19). In addition, we make sure that any file that is opened with superuser privilege is closed before exec (rule 21).
- finally, we model certain site-specific policies that override any access policies configured into ftpd. These policies are captured by rules 22 through 25.

3.3 Discussion

We make the following observations about the specification for `ftpd`.

- In order to reduce clutter, we have deliberately abbreviated some of the lists (e.g., `ftpInvalidUsers`), while leaving out definitions of some abstract events (e.g., `wrOpen`) in the specification for `ftpd`.
- Typical specifications need not be as comprehensive as for ftpd – we have made it comprehensive in order to better illustrate what sorts of properties can be captured in our language.
- The specification was developed using the principle of least privilege, without really paying attention to known vulnerabilities. Nevertheless, it does address most known ftp vulnerabilities (many of which have since been fixed) such as FTP bounce (rule 25), race conditions in signal handling (rules 20, 19) and site-exec (rule 24) [CERT].
- Availability of fast matching algorithms described in the subsequent sections enables us to focus on developing these rules independent of each other, as opposed to worrying about how they may be modified to enable more optimal checking, e.g., rules 11, 15, 16, and 23 have prefixes in their pattern component that are similar, and we can in fact combine them into a single rule. But this will lead to a specification that is much less clear, so we avoid this. Since the matching algorithms give us the benefit of such combination for free, there is no cost to pay for the clarity of specifications.
- We can develop a more concise language that gives us the ability to specify dependencies among events e_1 and e_2 more directly, rather than using a pattern such as $begin(); (!e_1)*; e_2$. However, the focus here is on a core language that has the necessary expressive power.
- The examples illustrate that the expressive power of REE is far beyond that of regular expressions. For instance, rules 20 and 21 capture associations among events that are beyond what can be captured even by context-free grammars. (The associations are similar to checking that a variable is defined before use, and can be captured by attribute grammars.)

4 Runtime Model

Our runtime model for the detection engine is based on extended finite-state automata (EFSA). EFSA are simply standard finite state automaton that are augmented with the ability to store values in a fixed number of *state variables.* Every transition in the EFSA is associated with an event, an enabling condition involving the event arguments and state variables, and a set of assignments to state variables. The final states of the EFSA may be annotated with actions, which, in our system, will correspond to the response actions given in our specifications. For a transition to be taken, the associated event must

[3]We note that it may be hard to obtain a complete list of the system calls in either of these cases.

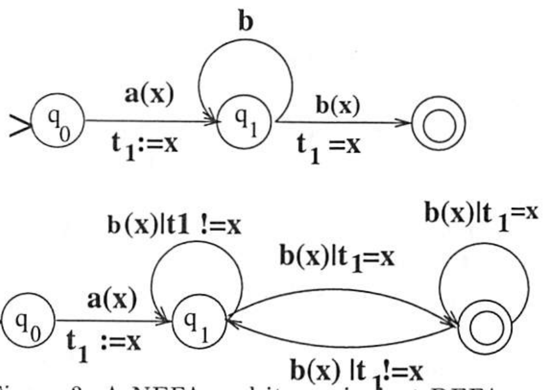


Figure 3: A NEFA and its equivalent DEFA

occur and the enabling condition must hold. When the transition is taken, the assignments associated with the transition are performed.

An EFSA is normally nondeterministic. The notion of acceptance by a nondeterministic EFSA (abbreviated as NEFA) is similar to that of an NFA: a NEFA accepts an event history $e_1 e_2 \cdots e_n$ if there is a sequence of states $s_0, s_1, \ldots s_n$ such that s_0 is the start state, s_n is a final state, and $\forall\ 1 \leq i \leq n$, there exists a transition from s_{i-1} to s_i that can be taken on e_i. A deterministic EFSA (DEFA for short) is an EFSA in which at most one of the transitions is enabled in any state of the EFSA. A NEFA for the pattern $a(x); b*; b(x)$ is shown in Figure 3. The equivalent DEFA is also shown in the same figure. We summarize some of the key properties of REE and EFSA, stated without proof. Let the REE be of size N, and use k variables $X_1, ..., X_k$ where X_i can assume n_i distinct values, for $1 \leq i \leq k$.

- there exists a NEFA of size N corresponding to the REE and it uses state variables $X_1, ..., X_k$
- this NEFA can be simulated with at most $O(N * n_1 * n_2 * \cdots * n_k)$ cost per event at runtime
- every NEFA can be transformed into an equivalent DEFA with the same set of state variables
- in the worst-case, the smallest DEFA for the above REE will have $2^{N*n_1*n_2*\cdots*n_k}$ states

The first and third properties are among those that carry over from regular expressions to REE. The second property shows that the overhead for simulating NEFA at runtime can be significantly higher than that for simulating NFA, the latter being just $O(N)$. Similarly, the fourth property shows that the size explosion due to NEFA to DEFA conversion can be significantly worse than NFA to DFA conversion, the latter explosion being limited to $O(2^N)$ in the worst case, and much smaller in practice. In fact, the explosion for NEFA to DEFA construction is unacceptably large, making such conversion impractical. Before we proceed to tackle this problem in the next section, we first develop an algorithm for using NEFA for matching event histories at runtime.

```
procedure $NEFAtrans$(HashTable $curStats$, Event $e$)
    foreach state $S \equiv (c, v_1, ..., v_n) \in curStates$ do
      delete $S$ from $curStates$
      foreach transition $T$ from $c$ to a state $c'$ enabled
          on $e$ with value $v_i$ for state variable $x_i$ do
        Let $v'_1, ..., v'_n$ be the new values of state
          variables after making the assignments in $T$
        foreach $S' \in epsilon((c', v'_1, ..., v'_n))$ that is
            not already in $curStates$ do
          add $S'$ to $curStates$
      end
    end
end

procedure $NEFAsim$(EventHistory $H$)
    Let $S^i \equiv (c^i, v^i_0, ..., v^i_n)$ denote the initial state
    Let $H$ be of the form $e_1, e_2, ..., e_m$
    Initialize a HashTable $curStates$ to contain $S^i$
    for $1 \leq j \leq m$ do
      $NEFAtrans(curStates, e_j)$
    end
end
```

The algorithm uses a hash table to store all of the possible states in which the NEFA could be in. On receipt of an event, we compute the new states of the NEFA by making a transition from each of the current states. A state of the NEFA is represented by a tuple $(c, v_1, ..., v_n)$ where c denotes the control state of the NEFA and $v_1, ..., v_n$ denote the values of the state variables. The algorithm uses a function *epsilon* to compute the set of states reachable from a state S while only following ϵ-transitions.

We remark that our NEFA model has some similarities with the colored Petrinet model used in [Kumar95]. In particular, the notion of a token in their Petrinet corresponds to our notion of a NEFA state. Their semantics of matching differs from ours in some ways, and as a result, their algorithm for simulating a nondeterministic Petrinet results in cloning of tokens at each step of the simulation algorithm. This can result in unbounded increase in the number of tokens, whereas the number of distinct states of the NEFA is bounded as discussed above.

5 Translation of REE to NEFA

We propose an algorithm that achieves a balance between the space explosion implied by DEFA and the runtime inefficiency imposed by NEFA in this section. The key point is that the explosion due to state variables ($2^{n_1 * n_2 * \cdots * n_k}$) is unacceptable, whereas the explosion due to the size of REE (2^N factor) does not usually pose a problem in practice. Our algorithm is thus geared towards avoiding size explosion due to state variables. This is achieved by permitting nondeterminism in the automata on a subset of transitions where state variables are being assigned new values. This approach results in an EFSA that is deterministic for a subclass of REEs called REE(0) patterns. For non-REE(0) patterns, the EFSA produced may not be deterministic, but our experimental results show that the performance of the algorithm is very good even for such patterns. Regardless of the nature of the patterns, the EFSA size is $O(2^N)$. We present an overview of our algorithm here, leaving out the details due to space constraints. (See [SU99] for a more complete description.)

5.1 Translation Algorithm

Our algorithm has an initial preprocessing phase that includes the following steps:

- strip the leading *begin* from patterns; if a pattern does not start with a *begin*, prefix *any**, so that the augmented pattern will match an event history iff the original pattern matched a suffix of the history.
- introduce an end-marker # to every pattern, and then combine them into one pattern using ||.
- express realtime or atomicity using *, ; , || and constraints on argument values
- rename event arguments so that ith argument to any event is named \$i. For arguments that need to be remembered for later, we assign their values to state variables using notation $/t_j = \$i$. A pattern that does not use a state variable after this step is known as an REE(0) pattern.
- number the positions in the REE from left to right, indicating the positions as superscripts

Given the patterns $begin(); (a(x,x)||b); (a||b)*; b(x)$ and $begin(); (a(y,3)||c(y)); (a||c)*; c(y)$, the preprocessing steps result in the following pattern:

$$(a^1|\$1 = \$2)/t_1 := \$1; (a^2|b^3)*; b^4/\$1 = t_1; \#^5 \;\; || \\ (a^6|\$2 = 3||c^7)/t_2 := \$1; (a^8||c^9)*; c^{10}|\$1 = t_2; \#^{11}$$

The purpose of numbering the REE positions is similar to that of earlier algorithms for constructing DFA from regular expressions [Aho90, MY60]. Each state S in the NEFA is associated with a subset P_S of positions in the REE as follows: a position $p \in P_S$ iff there is a path from the start state of the NEFA to S that matches the prefix of the REE up to position p. Intuitively, P_S denotes that set of positions in the REE up to which we may have matched an input event history that took the NEFA from its start state to the state S.

The construction of the NEFA for an REE R begins with the start state S_0. The positions P_{S_0} associated with the start state is defined as $first(R)$, where $first$ is the function defined below.

$$\begin{array}{rcll} first(e|C) & = & \{pos(e|C)\} & \\ first(A;B) & = & first(A) \cup first(B) & \text{if } empty(A) \\ first(A;B) & = & first(A) & \text{otherwise} \\ first(A|B) & = & first(A) \cup first(B) & \\ first(A*) & = & first(A) & \end{array}$$

Here, the notation $pos(e)$ denotes the position number associated with the event e. $empty(A)$ is true if A matches the empty string. The $first$ of the example REE would be the positions $\{1, 6, 7\}$.

The next step in construction is to identify the distinct events $e_1, \ldots, e_m$ that occur at one of the positions in P_S. We create m transitions from S to *dummy* states as shown with dotted lines in Figure 4. The set of positions P_{S_i} associated with each of these dummy states is given by:

$$P_{S_i} = \{j \in P_S | e_i \text{ occurs at } j \text{ in } R\}$$

Let C_j be the condition associated with a position

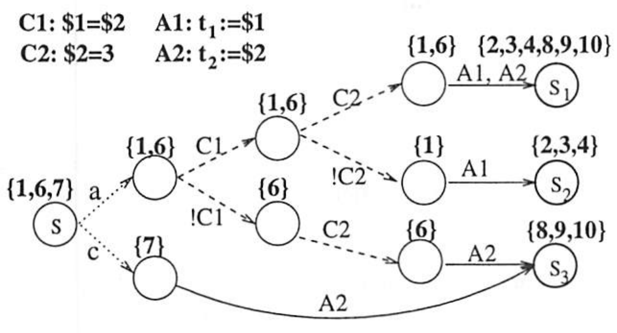


Figure 4: Partially constructed NEFA

$j \in P_{S_i}$. As the next step, we select a $k \in P_{S_i}$ and create two transitions from S_i to dummy states S_{i1} and S_{i2}, the first one to be taken when C_k holds and the second to be taken otherwise. $P_{S_{i1}}$ and $P_{S_{i2}}$ are given by:

$$P_{S_{i1}} = \{j \in P_{S_i} | C_j \wedge C_k \neq false\}$$
$$P_{S_{i2}} = \{j \in P_{S_i} | C_j \wedge \neg C_k \neq false\}$$

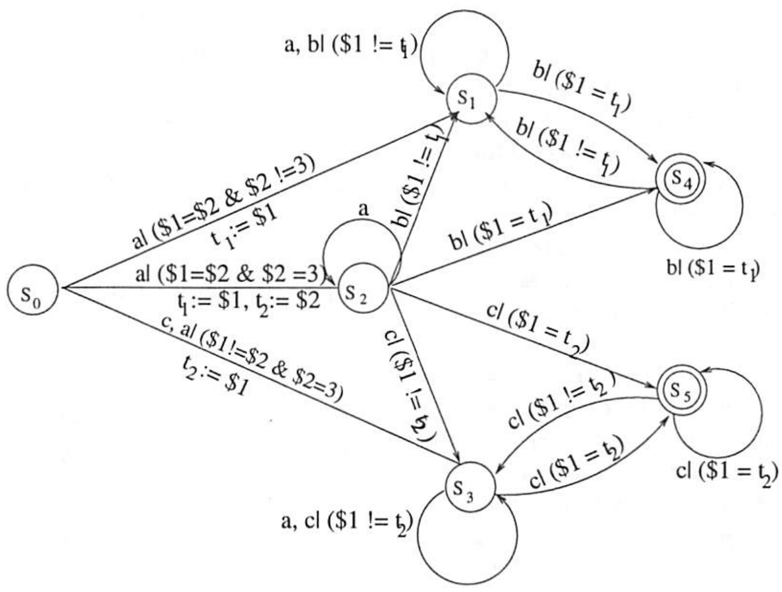


Figure 5: NEFA (also a DEFA) for the example patterns

We note that the correctness of the algorithm is not affected even if we included a j in $P_{S_{i1}}$ such that $C_j \wedge C_k = false$, but the size of NEFA may increase. A similar comment applies to $P_{S_{i2}}$ as well. (The size increase will likely be reversed by an optimizing compiler that may be used to compile the code generated by our NEFA construction algorithm.) This process is repeated at S_{i1} and S_{i2} recursively, while ensuring that each condition is tested at most once. The dashed lines in Figure 4 show these transitions.

When we reach a dummy state S_d where all conditions have been tested, we are ready to create new (non-dummy) states. Let $A_1, ..., A_r$ be the distinct assignment actions among the positions P_{S_d}. Then we create a total of r states $S'_1, ..., S'_r$, with ϵ-transitions from S_d to S'_k labeled with assignment actions A_n, for $1 \leq n \leq r$. $P_{S'_k}$ is determined as follows. Let P_k denote the subset of P_{S_d} where assignment actions A_k appear. Then $P_{S'_k} = \cup_{p \in P_k} follow(p)$, where $follow$ is defined by:

$follow(p) \supseteq first(Q)$, if p is rightmost for a subexpression $Q*$ in R

$follow(p) \supseteq first(Q)$, if p is rightmost within Q' where $Q'; Q$ is in R

$follow(p) \supseteq follow(q)$, if p and q are rightmost in Q and Q', where $Q || Q'$ is in R, or $Q; Q'$ is in R and $empty(Q')$

A position p' is included in $follow(p)$ only if required by the conditions above. We can reduce the number of ϵ transitions by merging states corresponding to assignments from different patterns. (The set of positions of the merged state is given by the union of these sets for the original states.) We can also avoid creating a new (non-dummy) state S'_n if $P_{S'_n}$ is identical to $P_{S''}$ for another state S'' in the NEFA. Figure 4 shows the partially constructed NEFA after these optimizations.

After the above algorithm, a post-processing phase merges sequences of dummy transitions into a single one, eliminating the dummy states in the process. It also marks any state that includes an end-marker position as a final state for the corresponding pattern. The NEFA constructed by the algorithm for our example is shown in Figure 5.

5.2 Properties of NEFA

Matching REE(0) Patterns As mentioned earlier, $REE(0)$ patterns are those that cannot refer to values of event arguments except immediately after the occurrence of the event. With our variable renaming scheme in place, this means that there would be no need for state variables, and hence no assignments. Note that the only source of nondeterminism in the above algorithm arises because of ϵ-transitions associated with each group of assignments. Therefore, the NEFA generated is a DEFA for $REE(0)$ languages.

Size It is easy to see from our algorithm that after the elimination of dummy states, there can be at most 2^n states, where n is the size of the REE, because each state is associated with a subset of positions in the REE, and there are at most 2^n such subsets. It is interesting to note that although we are dealing with a more complex language than RE, our bound for number of states is the same as the corresponding bound for RE.

To get an idea of the size of the NEFA, one has to consider the states as well as the space required for storing the transitions. If $maxp$ denotes the maximum number of positions where the same event occurs in an REE, then we can show that the total space requirements for the states and transitions is bounded by $O(2^{n+maxp})$.

We note that as in the case of the DFA construction algorithm from [Aho90], the worst case space requirements do not reflect the space usage in practice. Often, as in the case of the above example and in the examples studied Section 6.3, we end up with an automaton whose size is much smaller than the exponential upper bound.

6 Implementation and Performance

Our system implementation consists of a compiler and a runtime system.

6.1 Compiler

The front-end of the compiler is responsible for parsing a specification. Its implementation is routine, based on standard compiler construction tools Flex and Bison. Type-checking is performed subsequently, and then we translate the pattern components of our specifications into a NEFA using the algorithm described in the previous section. The reaction components of the rules are attached to those states that accept the pattern corresponding to the respective rules.

The NEFA is then translated into a C++ class *NEFA* as follows. At runtime, the state of the NEFA is maintained in a structure called *FSM*, which stores the control state as well as the state variables. To support nondeterminism, multiple instances of an FSM can be created using a *clone* function that makes identical copies of an FSM. At any point during runtime, zero, one or more copies of the FSM may be active.

The transitions of the NEFA are captured in the C++ code as follows. A system call *sc* is delivered to the NEFA class by invocation of a member function on the NEFA class with the name *sc*. The code for this class sequences through the list of FSM's that are currently active. For each FSM, its state is updated to reflect a transition that it would have made on the event *sc* and its arguments. If multiple transitions are possible, suitable number of copies of the FSM are made and each copy follows one of the transitions.

6.2 Runtime System

The output code produced by the compiler is linked with a runtime support system that provides the infrastructure for intercepting and delivering system calls to the detection engine. Events are delivered by invoking the corresponding member functions on the NEFA class described above. The runtime system also provides the functions needed by the detection engine to alter the behavior of system calls. With regard to data access, it provides support functions to access system call argument values.

The runtime system has been implemented by modifying the system call interface within the operating system. We do not provide a detailed explanation of the runtime system here, as our focus in this paper is on efficient runtime matching operations. Our results indicate that the overhead for interception is more or less constant per system call, and adds about 30% overhead to the cost of a system call [Bow99]. However, since system calls account for only a small fraction of the CPU time used by an application, the increase in CPU time (counted as the sum of user and system time on UNIX) is in the range of 2% to 30%. Most applications, especially those that are CPU-intensive, impose overheads at the bottom of this range, while I/O intensive applications such as `tar` and `ftp` lead to higher overheads. [FBF99], which employs an interception mechanism similar to ours, reports overheads in the range of 2% to 4% for a different set of applications, namely, `gcc` and `httpd`.

We remark that the overheads for system call interception are nontrivial. In fact, our fast matching algorithm reduces the checking time to the point where system call interception accounts for the the dominant portion of the total overhead imposed by our approach. In particular, this implies that there is no penalty to be paid due to the use of sophisticated pattern-matching based approach such as ours over a method that uses simpler rules that are triggered on individual system calls.

TestCase	Time to run(s)	Time to match all sysCalls (s)	Overhead
ftpd	2.2s	0.03	1.5%
telnetd	3.1s	0.04	1.3%
httpd	5.8	0.09	1.5%

Figure 6: Overhead due to system call pattern matching. Results taken on 350MHz Pentium II Linux PC with 128MB memory and 8GB EIDE disk.

6.3 Performance Results

We studied the performance of our system with three server programs, namely, `ftpd`, `telnetd` and `httpd`. The specification for `ftpd` was as described earlier. The specification for `telnetd` and `httpd` are not shown, but are comparable in size and complexity to that of `ftpd`. Our experiments were designed to evaluate the runtime overhead for monitoring (both in terms of time and space) and the size of the pattern-matching automata. In measuring the runtime overhead, we omitted the cost of intercepting system calls since our primary interest is in evaluating the performance of the pattern-matching algorithms. Moreover, the overhead for interception remains essentially a constant independent of the size of the patterns or length of the execution trace.

6.3.1 Timing Results

The best indicator of runtime overheads is the fact that only two of all the patterns in these three specifications were non-REE(0). This implies that the runtime matching effort consists of performing a transition on a deterministic automaton, which is about as expensive as the cost of the tests involved in the applicable transitions. As compared to the cost of a typical system call, this is indeed very low. The non-REE(0) patterns did not contribute much to the overhead either, as they led to very few cases where cloning was required. Specifically, the number of clones active at any point at runtime was less than five, with each FSM requiring several bytes of storage. Thus the runtime storage requirements were very low. Simulating such a small number of FSMs at runtime is also very fast.

Figure 6 summarizes our experimental results. They show that in fact, the overheads due to the monitoring are almost imperceptible for all three applications. The runtime data storage requirements for the FSM are too small to be measured.

Figure 7 shows the overhead for matching each system call, averaged across the three programs. The graph indicates a slight increase with increase in number of system calls, which is to be expected due to the presence of non-REE(0) rules. However, the rate of increase is extremely small, at about 5% when the number of system calls has increased by about 2000%. Thus, it is meaningful to talk about overhead as a percentage of the total runtime, as was done in Table 6. To obtain the results in the table and the figure, we exercised the three programs with a random combinations of valid commands of different lengths.

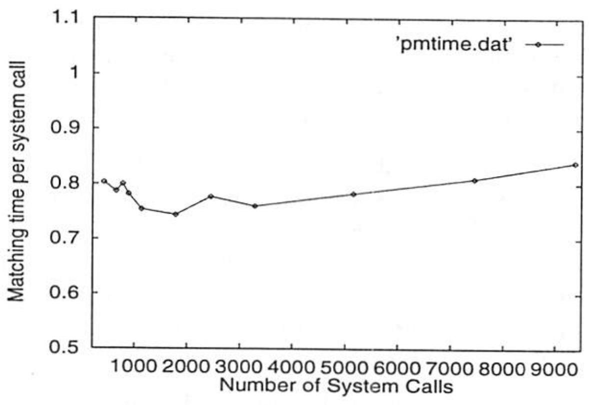


Figure 7: Matching time per system call (in microseconds).

6.3.2 Automaton Size

To evaluate the increase in size of the automaton when the number or complexity of patterns is increased, we have plotted the automaton size as a function of the size of the patterns. The size measure we use is given by the number of REE events and operators, taken over all of the patterns. Event

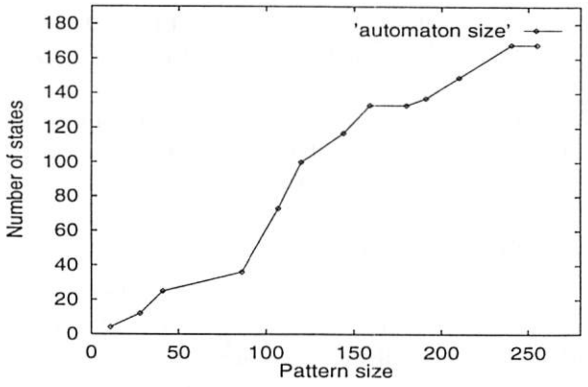


Figure 8: Increase in automaton size with specification size.

arguments and conditions are not included, as the number of states is not very much affected by their presence or absence. The REE patterns of interest were those corresponding to our three benchmark programs, `ftpd`, `telnetd` and `httpd`. Randomly chosen subsets of these patterns were compiled and the corresponding size of the automaton was identified. These were then plotted as shown in Figure 8.

Although the worst-case size is exponential in the size of REE, we find that in practice, the size increases more or less linearly with the total REE size. Although it is not readily apparent from the graph, the rate of increase in number of states tended to decrease after a while in our examples. The reason for this is that existing patterns had already created many states that could already represent some of the stages of matching the new pattern, and thus only fewer additional states were required. However, we must exercise caution in generalizing from these three programs, as the relationship between pattern size and automaton size can vary greatly from one set of patterns to another.

7 Related Work

7.1 Intrusion Detection

Techniques for prevention of intrusions draw on previous research on (post-attack) intrusion detection. Intrusion detection techniques can be broadly divided into *misuse* detection [PK92, Kumar95], *anomaly* detection [ALJTV95, FHS97, GSS99], and *specification-based* detection [Ko96, SBS99].

Among misuse-based approaches, a state-transition diagram based approach is used in [PK92] to capture signatures of intrusions. [Kumar95] uses colored petri nets to specify intrusive activity. This language is more expressive than ours in some ways (e.g., ability to capture occurrence of two concurrent sequences of actions), and less expressive in some other ways (e.g., ability to capture atomic sequences or the occurrence of one event immediately following another). Nevertheless, most intrusion signatures expressed in [Kumar95] can be easily captured in our language as well and hence our compilation techniques are applicable to their approach.

Among anomaly detection approaches, one of the first works based on program behaviors (as opposed to user behaviors) was that of [FHS97]. Recently, these results have been improved by [GSS99] using a neural network based approach that produces very accurate anomaly detectors. All these approaches deal only with system call names, not with arguments. This simplifies the problem of *learning* normal behaviors of processes, which is the main focus of their work. However, for intrusion prevention or confinement, argument values are indeed important, e.g., we cannot otherwise distinguish an action to write a log file from one to modify the `/etc/passwd` file. Thus a language like REE is more appropriate in this context.

A specification-based approach achieves the accuracy of misuse detection, while addressing one of its deficiencies, namely, the inability to deal with unknown intrusions. It was first proposed in [Ko96]. They use a pattern language based on context-free grammars extended with variables, and formulate the intrusion detection problem as one of parsing the audit logs with respect to these grammars. In contrast, our language is based on an extension of regular languages with variables. While context-free languages are more expressive than regular languages, this is not necessarily true when variables have been added to these languages. On the other hand, a regular language based formulation lends itself more readily to an automaton based pattern-matching approach that can be implemented efficiently.

7.2 Preventive Approaches

Some of the preventive approaches are based on protecting systems against underlying software errors that are exploited by attackers. For instance, malicious code detection techniques [BD96, GM96, LLO95] employ program analysis techniques to detect security-related errors in the source code. Similarly, [CPMWBBGWZ98] developed compilation techniques that add additional checks into the code generated to identify (and prevent) attacks that exploit buffer overflows to overwrite the return addresses stored on a process stack.

Interception of system calls, followed by interposition of arbitrary code at this point, has been proposed by many researchers as a way to confine applications. The Janus system [GWTB96] incorporates a user-level implementation of system call interception. It is aimed at confining helper applications (such as those launched by web-browsers) so that they are restricted in their use of system calls. Their language is tailored to restrict individual system calls without any regard for the context in which they appear. This approach is well-suited for fine-grained access control and sandboxing. The kernel hypervisor [MLO97] approach is similar to the Janus approach, but is implemented within an operating system kernel using loadable modules. A more comprehensive set of system call interposition capabilities was developed in [GPRA98]. Their approach is geared for a broad range of applications aimed at augmenting software functionality in areas such as security and fault-tolerance. [FBF99] focuses on the related problem of developing languages customized for writing interposition code (also known as *wrapper* code), and runtime infrastructure for their installation and management. Unlike the preceding approaches, their language can more easily capture sequencing relationships among system calls. But they do not focus on pattern-based techniques for intrusion detection. Moreover, computational issues in efficient matching of system call sequences are not addressed.

7.3 Other Work

There has been a significant amount of earlier work in regular languages and pattern-matching. In particular, our algorithm for NEFA construction is based in part on some of the ideas developed in [Aho90, MY60] for direct construction of DFA from regular expressions. The language design has been influenced by previous pattern-matching based languages such as Lex and Awk [AWK88].

Regular languages and ω-regular languages, together with their automata-equivalents, have been used widely in formal specification and verification of concurrent systems [Kurshan94]. In the related area of program analysis, [OO90] develops an approach for expressing sequencing constraints as regular expressions, and compile-time checking of these constraints using dataflow analysis. [Schneider98] proposes to use Buchi automata for monitoring security properties of programs. The main difference between these works and ours is that they operate on regular or ω-regular languages, while our language is REE. Another difference is that in the context of verification, we are often interested in properties of infinite sequences, while our interest is in behaviors that manifest themselves in finite sequences.

Describing properties of event sequences is an important problem in building, prototyping, debugging and monitoring distributed systems. Languages such as CSP [Hoare78], LOTOS [BB89] and Esterel [BCG87] support very simple patterns, permitting no sequencing or closure operators. The task sequencing language developed in [LHMBH87] and its successor Rapide [LV95] support an expressive pattern language that is significantly more expressive than ours. Moreover, they support a partially ordered set model of event histories, whereas our current model is a linear sequence. However, it is not clear whether this language is amenable to efficient pattern matching at runtime, as they do not address the problem of automata-based techniques for efficient matching of these patterns.

8 Conclusions

In this paper we presented an approach for building survivable systems. Our approach is based on monitoring system calls made by processes (running on the system to be protected), comparing them against patterns characterizing normal (or abnormal) system call sequences, and initiating appropriate responses when (potentially) damaging system calls are made. These responses may preempt intrusion, or otherwise isolate and contain any damage. Since attacks can be prevented and/or contained, our approach can satisfactorily address intrusions arising due to software errors in otherwise trustworthy programs, as well as malicious programs (e.g., Trojan horses) from untrusted sources. Moreover, when new vulnerabilities (not protected by existing specifications) are identified, we can protect against them using appropriate specifications, instead of disabling vulnerable software until the vendor provides a patch.

One of the main challenges in making our approach practical is the ability to perform runtime prevention/detection that is fast enough to be included as part of processing every system call made by every process. We proposed a solution to this problem in this paper by developing an algorithm that compiles patterns in our specification language into an extended finite state automaton (EFSA). Our implementation results demonstrate that detection can be performed very fast (1 to 2% overhead to program execution time) using our approach. A key benefit of our approach (supported by theory as well as performance experiments) is that the detection time is insensitive to the complexity or number of patterns used in the specification. In most cases, our algorithm takes a constant time per system call intercepted, and uses a constant amount of storage. These advantages make our algorithm attractive for any other approach that performs intrusion detection by some form of pattern-matching.

We showed that for a class of patterns (specifically, REE(0) patterns) the automata constructed by our algorithm are deterministic. A more precise characterization is desirable, since in practice, our algorithm builds deterministic automata for a much larger class of patterns. Such a characterization may also enable us to restrict the specification language so that patterns that can lead to very large overheads (associated with simulating a large number of NEFA instances at runtime) can be avoided. Another area of future research is the application of techniques presented in this paper to the larger problems of distributed system monitoring and software debugging.

References

[Aho90] A.V. Aho, Algorithms for Finding Patterns in Strings, Handbook of Theoretical Computer Science Vol A, Elsevier Science Publishers B.V., 1990.

[AWK88] A.V. Aho, B.W. Kernighan, and P.J. Weinberger, The AWK Programming Language, Addison-Wesley, Reading, MA, 1988.

[ALJTV95] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes, Next-generation Intrusion Detection Expert System (NIDES): A Summary, SRI-CSL-95-07, SRI International, 1995.

[BCG87] G. Berry, P. Couronne and G. Gonthier, Synchronous Programming of Reactive Systems: An Introduction to Esterel, Technical Report 647, INRIA, Paris, 1987.

[BD96] M. Bishop, M. Dilger, Checking for Race Conditions in File Access. Computing Systems 9(2), 1996, pp. 131-152.

[BB89] T. Bolognesi and E. Brinksma, Introduction to the ISO Specification Language LOTOS, The Formal Description Technique LOTOS. Amsterdam: North-Holland, 1989.

[Bow99] T. Bowen et al, Operating System Support for Application-Specific Security, under review for Symposium on Operating Systems Principles, 1999.

[CERT] CERT Coordination Center Advisories 1988–1998, `http://www.cert.org/advisories/index.html`.

[CPMWBBGWZ98] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang, StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, 7th USENIX Security Symposium, 1998.

[Denning87] D. Denning, An Intrusion Detection Model, IEEE Trans. on Software Engineering, Feb 1987.

[FHS97] S. Forrest, S. Hofmeyr and A. Somayaji, Computer Immunology, Comm. of ACM 40(10), 1997.

[FHRS90] K. Fox, R. Henning, J. Reed and R. Simonian, A Neural Network Approach Towards Intrusion Detection, National Computer Security Conference, 1990.

[FBF99] T. Fraser, L. Badger, M. Feldman, Hardening COTS software with Generic Software Wrappers, IEEE Symposium on Security and Privacy, 1999.

[GPRA98] D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson, SLIC: An Extensibility System for Commodity Operating Systems, USENIX Annual Technical Conference, 1998.

[GM96] B. Guha and B. Mukherjee, Network Security via Reverse Engineering of TCP Code: Vulnerability Analysis and Proposed Solutions, Proc. of the IEEE Infocom, March 1996.

[GSS99] A.K. Ghosh, A. Schwartzbard and M. Schatz, Learning Program Behavior Profiles for Intrusion Detection, 1st USENIX Workshop on Intrusion Detection and Network Monitoring, 1999.

[GWTB96] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, A Secure Environment for Untrusted Helper Applications, USENIX Security Symposium, 1996.

[Hoare78] C. Hoare, Communicating Sequential Processes, Comm. of the ACM, 21(8), 1978.

[Jones93] M. Jones, Interposition Agents: Transparently Interposing User Code at the System Interface, 14th ACM Symposium on Operating Systems Principles, December 1993

[Ko96] C. Ko, Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach, Ph.D. Thesis, Dept. Computer Science, University of California at Davis, 1996.

[Kumar95] S.Kumar, Classification and Detection of Computer Intrusions, Ph.D Dissertation, Department of Computer Science, Purdue University, 1995.

[Kurshan94] R. Kurshan, Computer Aided Verification of Coordinating Processes: The Automata-Theoretic Approach, Princeton University Press, Princeton, NJ, 1994.

[LLO95] R.W. Lo, K.N. Levitt, R.A. Olsson, MCF: a Malicious Code Filter, Computers and Security, Vol.14, No.6, 1995.

[LV95] D. Luckham and J. Vera, An Event-Based Architecture Definition Language, IEEE Transactions on Software Engineering, 21(9), 1995.

[LHMBH87] D. Luckham, D. Helmbold, S. Meldal, D. Bryan, and M. Haberler, Task Sequencing Language for Specifying Distributed Ada Systems: TSL-1, PARLE: Conf. on Parallel Architectures and Languages, LNCS 259-2, 1987.

[Lunt92] T. Lunt et al, A Real-Time Intrusion Detection Expert System (IDES) - Final Report, SRI-CSL-92-05, SRI International, 1992.

[MY60] R. McNaughton and H. Yamada, Regular expressions and state graphs for automata, IRE Trans. on Electronic Comput., EC-9(1), 1960.

[MLO97] T. Mitchem, R. Lu, R. O'Brien, Using Kernel Hypervisors to Secure Applications, Annual Computer Security Application Conference, December 1997.

[OO90] K. Olender and L. Osterweil, Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation, IEEE Transactions on Software Engineering, 16(3), 1990.

[PK92] P. Porras and R. Kemmerer, Penetration State Transition Analysis:A Rule based Intrusion Detection Approach, Eighth Annual Computer Security Applications Conference, 1992.

[Schneider98] F. Schneider, Enforceable Security Policies, TR 98-1664, Department of Computer Science, Cornell University, Ithaca, NY, 1998.

[SCS98] R. Sekar, Y. Cai and M. Segal, A Specification-Based Approach for Building Survivable Systems, NISSC, October 1998.

[SBS99] R. Sekar, T. Bowen and M. Segal, On Preventing Intrusions by Process Behavior Monitoring, USENIX Intrusion Detection Workshop, 1999.

[SU99] R. Sekar and P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications, Technical Report 99-03, Department of Computer Science, Iowa State University, Ames, IA 50014.

Building Intrusion Tolerant Applications[*]

Thomas Wu
tjw@cs.stanford.edu

Michael Malkin
mikeym@stanford.edu

Dan Boneh[†]
dabo@cs.stanford.edu

Abstract

The ITTC project (Intrusion Tolerance via Threshold Cryptography) provides tools and an infrastructure for building intrusion tolerant applications. Rather than prevent intrusions or detect them after the fact, the ITTC system ensures that the compromise of a few system components does not compromise sensitive security information. To do so we protect cryptographic keys by distributing them across a few servers. The keys are never reconstructed at a single location. Our designs are intended to simplify the integration of ITTC into existing applications. We give examples of embedding ITTC into the Apache web server and into a Certification Authority (CA). Performance measurements on both the modified web server and the modified CA show that the architecture works and performs well.

1 Introduction

To combat intrusions into a networked system one often installs intrusion detection software to monitor system behavior. Whenever an "irregular" behavior is observed the software notifies an administrator. In this paper we study a complementary approach we call *intrusion tolerance.* Rather than prevent or detect intrusions after the fact, we provide tools that limit the amount of damage an intruder can cause. Our goal is to ensure that an attacker who penetrates a few system components cannot compromise total system security. To defeat our system an attacker must either penetrate multiple components in a short amount of time or frequently penetrate a certain component. Either way such large scale attacks are much easier to detect than single isolated attacks.

To describe our approach we consider a web server as an example. To enable secure connections to the server one often stores a secret key on the server. Typically the key is used during SSL session key negotiation. An attacker who penetrates the server can expose the private key and can then either masquerade as the server or eavesdrop on connections to the server. Hence, in the case of a bank's web server a single intrusion could result in a compromise of sensitive financial data. In contrast, our approach splits the web server into a small number of components so that a single intrusion does not expose any information about the server's key. Our approach can also be used to safeguard a Certificate Authority's (CA) private key. By splitting the CA into a number of components we can ensure that penetration of a small number of components does not compromise the CA's key.

We protect an application's private key (e.g. a web server or a CA) by sharing the key among a number of *share servers.* An attacker who breaks into a small number of share servers cannot expose the private key. Our main design principle is that long term security information should *never* be located in a single location. Hence, there is no single point of attack at which an attacker can expose critical security information (such as a CA's private key). Shamir's secret sharing scheme [14] is a classic approach for distributing private keys across several sites. Unfortunately, to use the shared key one must reconstruct it at a single location. Thus, secret sharing is inappropriate for our purposes. Instead, we use techniques of Threshold Cryptography [9] to distribute an application's private key among several share servers so that the key can be used without ever reconstructing it.

Our system is designed to be easy to embed into existing applications. For example, by embedding our system into the Apache web server we built a web server whose private key is split across a number of share servers and is never reconstructed in a single location. Whenever the web server receives a request for a secure connection it interacts with the share servers to apply its private key. As we shall

[*]Project URL: `http://www.stanford.edu/~dabo/ITTC`
[†]Supported by DARPA contract #F30602-97-C-0326.

see in Section 7 the performance penalty for doing so is small compared to the total time to establish an SSL connection. Consider again the example of a bank's system. By placing the share servers in a secure subnet and ensuring that they only accept connections from the bank's web servers one can strengthen the security of the bank's private key. Furthermore, our share servers can service multiple web servers. Hence, a bank maintaining dozens of web servers need not store a private key on each of them. Instead, the private keys can be stored (in shared form) on the share servers. This way a small number of break-ins will not compromise any of the private keys.

As an additional benefit, our system provides high availability of private keys. Even if a few of the share servers crash (by accident or as a result of an attack) and all data stored them is lost, the remaining active share servers automatically compensate for the corrupt ones. Furthermore, the system can recover by redistributing valid shares to the corrupted share servers.

The paper describes the design and implementation of our system including the prototype applications we built. We provide detailed performance measurements on both the modified web server and the modified CA to show that the architecture works and performs well.

2 System components

To protect private keys used by applications such as a web server or a CA, our system distributes shares of private keys among a number of share servers. In this section we describe the system components. We refer to applications using the share servers as *clients*. Web servers and CA's are example clients. Figure 1 illustrates the interaction between the various components.

Share server The share servers are implemented as daemons running on different machines. They hold shares of the private keys belonging to the clients they serve. These shares reveal no information about the client's private key. A share server can manage multiple keys and serve a large number of clients. We envision the total number of share servers being less than ten. Clearly it is desirable that an attacker not be able to compromise the share servers. However, the intrusion tolerant design of our system ensures that even if a few of the share servers are penetrated and the secrets stored on them are exposed or corrupted there is no compromise to overall system security. In other words, an attacker learns nothing from penetrating a few of the share servers. The system identifies corrupt share servers and can be instructed to take appropriate action to refresh the secrets stored on them.

Client. A client is any application that makes use of our client side Threshold LiBrary (the TLB). When a client connects to the share servers it first authenticates itself as an authorized client. It then interacts with the share servers to sign a message or decrypt a given ciphertext using the shared private key stored on the share servers. One concern is that an attacker can penetrate a client and expose the client's authentication key. The attacker can then masquerade as the client and fool the share server into applying a shared key. In Section 5.2 we describe a mechanism for detecting such an attack. At any rate, the attacker cannot expose the shared private key (as is the case when the private key is stored on the client).

Administrator. We provide a central administration utility to administer all the share servers. The administrator utility manages the various keys stored on the share servers. It can shutdown or suspend the servers if necessary, or instruct the servers to take appropriate action if some of the servers have been penetrated. The utility is provided for convenience. If for some reason central administration is not desired the share servers can be controlled locally.

Monitor. We built a monitor utility for testing and demonstrating our system. The monitor is a single daemon that collects information from all key servers and clients. Essentially, the share servers and clients send data to the monitor (via UDP) telling it what they are doing at any given moment. For instance, a client may tell the monitor that it is asking servers number 1, 2 and 4 to apply key number 23 to sign a message. A moment later share servers 1, 2 and 4 will each tell the monitor that they are applying shares of key number 23 to generate a signature, and so on. The monitor collects all this information and sends it to a Java applet that displays it in a human readable form. During actual deployment the monitor can be easily disabled by setting the appropriate switch in the system's configuration file. The administrator can still monitor system behavior by viewing each server's log file.

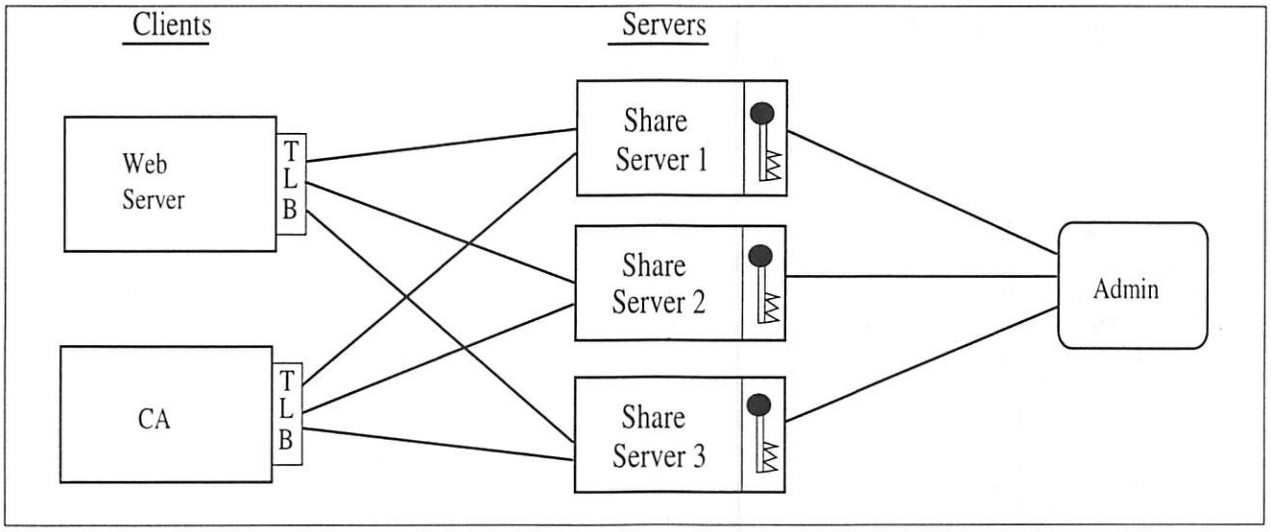


Figure 1: System components

2.1 Interaction between components

Each share server manages shares of the private key belonging to its clients. These shares are stored on disk encrypted using the administrator's passphrase. When the share server is started (possibly as part of the system's bootup script) it enters *suspended* mode. In suspended mode the share server refuses to accept any connections except a connection from the administrator. When all share servers are started, the admin program can be used to activate the servers by sending the passphrase to all of them at once. Connections between the admin program and share servers are always protected by SSL using mutual authentication. Using a hash of the administrator's passphrase, each server loads all shares available to it and enters *active* mode. The server is then ready to serve its clients.

In active mode a share server can accept connections from its clients and the administrator. The first step in any connection to a share server is mutual authentication and key exchange using SSL. The second step is an integrity check of the peer as described in Section 5.2. After a secure link is established, requests to the share server use the following protocol:

Request:	COMMAND ⟨*opcode*⟩ ⟨*data*⟩
Response:	RESPONSE ⟨*status*⟩ ⟨*data*⟩

The response status is either (1) OK indicating a successful completion, (2) SUSPENDED indicating that the share server is in suspended mode, or (3) a detailed error-code.

To understand the interaction between the system components we describe some of the opcodes supported by the share servers.

Connection from client. Each connection from a client causes the share servers to create a new thread (up to a specified maximum number of threads). These threads last as long as the client keeps open the connection to the share server, reducing the overhead of repeatedly opening and closing connections. A client can send a number of opcodes instructing the servers to apply a shared key. The interaction that takes place as a result of these commands is described in Section 4.

SIGN. Instructs the share server to apply its share of the private key for the purpose of generating a signature. The parameters include a digest of the message to be signed. We note that prior to taking part in the signing protocol described in Section 4 the key-servers verify that the message to be signed follows the PKCS1 format [11].

DECRYPT. Instructs the share server to apply its share of the private key for the purpose of decrypting a ciphertext.

HASH-SIGN. Instructs the share server to apply its share of the private key for the purpose of generating a hash of a signature. The purpose of this opcode is explained in the next section.

Connection from the admin. Some of the commands from the administrator include:

SHUTDOWN, SUSPEND, ACTIVATE. Manage share server modes. Shutdown causes all share server daemons to exit.

GENKEY. The admin can instruct the share servers to generate a new shared key. In our system private keys are *never* constructed in a single location. When an RSA key is generated it is generated in shared form. This uses a protocol due to Boneh and Franklin [3]. We use the implementation of shared RSA key generation described in [10]. The shares of the new private RSA key are stored on the share servers (encrypted with the admin password). The new public key is sent to the admin and is saved on the admin's machine. Authorized clients can then connect to the share servers to generate signatures or decrypt incoming messages using the new key.

A new public key is usually certified by a CA before it can be properly used. To obtain a public key certificate for the new key the admin must generate a *certificate request* to be sent to a CA. The format of a certificate request (specified in PKCS10) includes a *self signature* on the request. In other words, the newly generated *private* key must be used to sign the public key certificate request. The self signature ensures that the entity requesting the certificate has the corresponding private key. Unfortunately, in our case, the admin does not have the private key. In fact, no one has the private key — the private key is *always* stored in shared form on the share servers. Consequently, to generate the certificate request, the admin must connect to the share servers *as a client* and ask them to sign the request. Once the admin obtains the self signed request he can forward it to the CA. The CA will send back the certificate. We provide a utility that enables the admin to generate a self signed certificate request, as described in Section 6.1.

REFRESH. Suppose the administrator suspects the system is under attack and some share servers are compromised. In this case, the admin can instruct all share servers to *refresh* their shares of the private keys. Refreshing the shares does not change the private key. It simply generates a new independent *sharing* of the private key. Suppose an attacker obtains the shares of a private key stored on one of the servers. Once the admin refreshes the shares of the private key the information in the attacker's hands becomes useless.

Refreshing is also used when shares stored on a share server are corrupted or lost. Corruption or loss could happen due to a denial of service attack or a simple server crash. Refreshing the shares causes the uncorrupted servers to generate new valid shares for the corrupted server. Consequently, the system gracefully tolerates corruption or loss of a few shares.

3 Key management

We begin by explaining how one shares a private RSA key among a number of share server so that the key can be used without ever having to reconstruct it. We then describe the structure and PEM format used to store these shared keys.

3.1 Sharing RSA keys

Recall that an RSA private key consists of a modulus N and a secret exponent d. The modulus N is a product of two large primes, and d is a positive integer less than N. To decrypt a ciphertext C one computes $C^d \bmod N$. Similarly to sign a message digest M one computes $M^d \bmod N$. Hence, both operations require an exponentiation to the power d modulo N. Without the secret exponent d it is believed to be hard to either decrypt ciphertexts or generate signatures.

We show how a private RSA key can be broken up into a number of pieces (shares). Each share can be stored on a separate server and yet the private key can be used without having to reconstruct the secret. The basic idea, due to Frankel [6], is to pick random numbers d_1, d_2, d_3 in the range $[-N, N]$ so that $d_1 + d_2 + d_3 = d$. We then store share d_i on share server number i, for $i = 1, 2, 3$. Note that an attacker who breaks into any two of the three servers learns nothing about the private key d. All three servers must be compromised to obtain d.

When a client wishes to apply the key to sign a message M it sends M to all three servers. Each server applies its own share d_i to obtain $S_i = M^{d_i} \bmod N$ and sends the result S_i back to the client. The client obtains S_1, S_2, S_3 from the three servers. It multiplies them to obtain the signature

$$d = d_1 + d_2 + d_3$$
$$d = d_4 + d_5 + d_6$$

Server 1	Server 2	Server 3	Server 4
d_1	d_2	d_3	d_3
d_4	d_4	d_5	d_6

3-out-of-4 sharing

$$d = d_1 + d_2$$
$$d = d_3 + d_4$$

Server 1	Server 2	Server 3	Server 4
d_1	d_1	d_2	d_2
d_3	d_4	d_3	d_4

2-out-of-4 sharing

Figure 2: Additive sharings of a private RSA key d.

$S = S_1 \cdot S_2 \cdot S_3 \bmod N$. Since $d = d_1 + d_2 + d_3$ we have that $S = M^d \bmod N$ as required. Note that the private key d was never reconstructed in order to be used. In addition, there is no communication between the share servers. The only interaction is between the client and each of the servers.

Clearly this approach generalizes to distributing a private RSA key among k servers. Even if $k - 1$ of the shares are exposed, an attacker learns nothing about the private key d. Since all k share servers must be involved in applying to key we call this sharing a k-out-of-k sharing.

3.1.1 t-out-of-k sharing

There are a number of problems with the approach described above. Most importantly, it is not fault-tolerant. If one of the share servers crashes the entire system goes offline. If one of the share servers loses its share, the private key is lost forever. For this reason, we use a t-out-of-k sharing of the secret key; any t of the share servers can be used to apply the key. For instance, if we use a 3-out-of-4 sharing no harm is done if one of the share servers is taken offline. In addition, a break-in into any two servers reveals no information about the private key.

Typically a t-out-of-k sharing is achieved using Shamir's classic secret sharing [14]. Unfortunately, when using Shamir's secret sharing the keys must be reconstructed before they can be used. This is inappropriate for our purposes. Instead, we use a combinatorial construction to obtain a t-out-of-k sharing of d from several t-out-of-t sharings of d. We give two examples in Figure 2. In both examples all the d_i's are random integers in the range $[-N, N]$ satisfying the stated equalities. Each server stores multiple d_i's as indicated by the column corresponding to that server. Observe that in the example on the left, any three servers can apply the key while any two servers learn nothing about d. The example on the right is more fault tolerant, but compromising two servers reveals the key. A compromise of a single server reveals nothing about the key. More generally, we implemented an algorithm that constructs tables as above for a t-out-of-k sharing for any t and k. The algorithm is based on ideas from [1, 4].

When a client requests that the servers apply a private key, it specifies which coalition of t servers is used. Based on the coalition, each server locally decides which of its d_i's it will use and sends the resulting S_i back to the client. The client multiplies the t responses and obtains the signature S. The client then uses the public key to check the validity of S as a signature. This is done to ensure that all share servers responded properly. In Section 4 we explain how we deal with corrupt share servers that apply their private shares incorrectly.

3.2 Key structures and key storage

Our system manages three types of keys: (1) a standard RSA public key, (2) a private share stored on each share server, and (3) a public shared key stored on each client. The public key stored on the clients contains the public RSA key plus some additional public information. We describe each of these keys below.

The SSLeay package [16] supports reading and writing both public and private keys in PEM format. Our private shares and public shared keys are represented internally as extensions of the standard RSA key data structure. On disk, we support a PEM-encoded ASN.1 format similar to that used for RSA keys.

Public key This is a standard RSA public key made up of an RSA modulus N and a public exponent e. It is managed by the standard RSA functions provided in SSLeay. Note that

Figure 3: Shared key file formats.

type:	INT	INT	INT	INT	INT	INT	INT	INT	$\cdots$	INT
data:	version	N	e	k	t	w	d_1	d_2	$\cdots$	d_w

(a) Private key file format.

type:	INT	INT	INT	INT	INT	INT	INT	INT	INT	$\cdots$	INT
data:	version	N	e	k	t	g	u	g^{d_1}	g^{d_2}	$\cdots$	g^{d_u}

(b) Public shared key file format.

an outsider communicating with the clients is unaware that the corresponding private key is stored in shared form.

Private share The private share stored on each share server contains the modulus N and a number of private d_i's. For a t-out-of-k shared key the private share file format (as ASN.1) is shows in Figure 3a. Each share server is given w shares $d_1, \ldots, d_w$. Note that the private share file does not contain the optional values $d \bmod p-1$, $d \bmod q-1$, or $q^{-1} \bmod p$, normally used to optimize RSA computations, where $N = pq$. After all, none of the parties can construct these values.

Public shared key The format of the public key file stored on each client is shown in Figure 3b (all arithmetic is done modulo N). Here u is the total number of g^{d_i}'s stored on the client. The values g and $g^{d_i} \bmod N$ are used to detect incorrect (or possibly compromised) private share operations by the share servers, as discussed in the next section. For each share d_i on each share server, there is a g^{d_i} in the public shared key. For example, in the 3-out-of-4 sharing of the previous section the public shared key contains six entries, so $u = 6$.

4 Using a shared key

We are now ready to describe the interaction that takes place when a client requests the share servers to apply a private key to a message M. Throughout the interaction the client keeps two bit-vectors `offline` and `corrupt`, indicating which of the share servers are currently offline and which have been found to be corrupt. We assume the client wishes to use a t-out-of-k shared key.

Init The client sets `offline` and `corrupt` to zero.

Step 1: The client picks a *random* coalition of t servers (out of k) that are neither offline nor corrupt.

Step 2: The client sends the following message to each of the servers in the coalition:

COMMAND	SIGN or DECRYPT	M	
coalition	key-ID	corrupt	offline

where `key-ID` is the ID of the key the servers should apply. The `key-ID` is simply a 32 bit hash of N and e.

Step 3: Based on the coalition being used, each of the servers extract the appropriate d_i from its private key share of `key-ID`. It then locally computes $S_i = M^{d_i} \bmod N$ and send S_i back to the client. Note that if a signature is being requested, the client first verifies that the message M is in PKCS1 format, and rejects the request if not.

Step 4: The client collects all the S_i's and computes $S = \prod_{i=1}^{t} S_i \bmod N$. If some of the servers were found to be offline, the `offline` bit vector is updated and the process is restarted at Step 1.

Step 5: If $S^e = \pm M \bmod N$ the key was applied correctly and the process terminates.

Step 6: Otherwise, for each server in the coalition the client performs a zero-knowledge test to validate the server's response. The test is described in the next section. All servers that sent incorrect values are marked as corrupt and the process is restarted at Step 1.

If at Step 1 there are less than t servers that are neither corrupt nor offline the process fails. Note that the identity of corrupt share servers is sent to *all* share servers. This enables the servers to notify the administrator, who can take appropriate action to refresh the shares stored on the corrupt servers.

Load balancing When many different clients use the same share servers, the load on the servers may hurt overall system performance. Fortunately, our choice of a random coalition in Step 1 provides for *automatic load balancing* among the servers. When a 3-out-of-5 key is used each application of the key is likely to make use of a different set of three servers. Furthermore, by using a sufficiently low timeout period clients can avoid waiting for busy servers by switching to a different coalition. Due to the load balancing effect, a 3-out-of-5 sharing will result in a higher throughput than a 3-out-of-4 sharing.

4.1 Identification of corrupt servers

It remains to show how we identify corrupt servers in Step 6. This protocol is only used in the unlikely event one of the servers returns an incorrect response in Step 3. Recall that server i sent $S_i = M^{d_i} \bmod N$ back to the client where d_i is its share of the private key. By examining its own public shared key file the client obtains $V_i = g^{d_i} \bmod N$. To validate sever i's response the client does the following:

Step 6.1: The client picks random a and b in the range $[1, N]$ and computes $Z = M^a g^b \bmod N$.

Step 6.2: It sends the following to server i:

COMMAND	HASH-SIGN	Z	

	coalition	key-ID	corrupt	offline

Step 6.3: A legitimate server will respond with $A_i = H(Z^{d_i} \bmod N)$ where H is a cryptographic hash function. We use SHA-1.

Step 6.4: The client checks that $A_i = H[S_i^a V_i^b \bmod N]$. If not, the server is declared to be corrupt.

The above protocol is a simplification of a protocol from [7]. The following lemma shows that a share server who sends an invalid S_i will be caught with overwhelming probability. Furthermore, the protocol reveals no information about the server's secret shares. The lemma relies on the Small Order Assumption (SOA): there is no efficient algorithm that, given an RSA modulus $N = pq$, outputs an element x in $\mathbb{Z}_N$, where $x \neq \pm 1$, so that the order of x in $\mathbb{Z}_N$ is less than $N^{1/2}$.

Lemma 4.1 *Assuming SOA, a server who in step 2 sends any value other than* $\pm M^{d_i} \bmod N$ *will fool the above protocol with probability at most* $1/N^{1/2}$. *Furthermore, assuming the hash function used is a random function, the protocol is zero-knowledge (i.e. the client can simulate its interaction with the server).*

For a 1024 bit key, the probability that a server fools the client is less than $1/2^{512}$ — negligibly small. The hash function is used to prevent a malicious client from obtaining signatures on improperly formatted messages (i.e. not in PKCS1 format). For example, in a CA environment the DECRYPT command is disabled. Consequently, the servers can only be used to generate signatures and they only sign properly formatted PKCS1 messages. In this environment, the hash function prevents the client from using the validation test to obtain signatures on arbitrary messages.

We note that when $\gcd(p-1, q-1) = 2$ the SOA assumption is equivalent to assuming the hardness of factoring. However, since our servers cannot test this condition (no single server knows p and q) we use the SOA as the intractability assumption in Lemma 4.1.

5 Implementation details

5.1 Certificate formats

The ITTC system uses the Secure Socket Layer (SSL) to authenticate and encrypt all communication. The administrator issues a certificate to each entity (client, share server, and administrative utility) in the system, which must be in a recognized format. The CN (Common Name) field in each certificate contains a string that allows other parties to identify it as described in the following table:

Entity	ID String
Client	`[CLIENT` n`]`
Share Server	`[SERVER` n`]`
Administrator	`[ADMIN` n`]`

When establishing an SSL-secured connection, both peers must send their certificates to establish mutual authentication. Each peer verifies the certificate it receives and then parses it to extract the identifying string to ensure that the peer is authentic. If a certificate is not sent, if verification fails, or if the identity string does not match what was sent in the protocol, the connection fails.

5.2 Sequence numbers

Although client and server certificates thwart direct network snooping and impersonation attacks against the servers, the ITTC threat model requires the system to tolerate and resist successful attacks against individual clients and servers as well. If an attacker, for example, obtains the private key and certificate of a legitimate client, he could use it to obtain access to any shared keys that the client had access to. To limit the amount of damage under these circumstances, ITTC uses sequence numbers in the connection protocols to detect such compromises after the fact. Each client and server keeps a count of how many times it has been accessed by each other entity in the system, and all connections involve an exchange of sequence numbers to verify synchronization. An attacker who uses a stolen key and certificate will cause the sequence number at each server to be incremented without a corresponding change in the sequence number at the "victim" client's system. The next time that client attempts to access a server, the mismatch will be detected.

The actual multi-step sequence number exchange proceeds as follows. Assume that both client and server are currently synchronized at sequence number n initially:

1. The server sends n to the client and sets its own sequence number to $n + 1$.
2. The client expects n. It sends $n+1$ back to the server. It sets its own sequence number to $n+1$. (In this step, the client also accepts $n + 1$ from the server. If this happens, it sends $n + 2$ back to the server and sets its own sequence number to $n + 2$. The rest of the protocol proceeds with the client adding one to all its sequence numbers.)
3. The server expects $n + 1$. It sends $n + 2$ back to the client. It sets its own sequence number to $n + 2$.
4. The client expects $n + 2$. It sets its own sequence number to $n + 2$.

When the protocol completes successfully, both client and server hold sequence number $n + 2$. The advantage this protocol has over a simpler single-increment protocol is that it distinguishes between ordinary network failure and a genuine security breach. In a single-increment protocol, it is possible for a network failure to cause one party to increment its sequence number while the other does not. The off-by-one discrepancy in the sequence numbers would be interpreted as a security breach during subsequent interactions between these two parties. This double-increment protocol, on the other hand, increments sequence numbers by two for each successful interaction, while ensuring that the server's sequence number never drifts by more than one relative to the client's number. A network failure would result in at most an off-by-one discrepancy, which would then be resolved in step two of the protocol. A real security breach would cause the numbers to differ by two or more, presumably triggering a legitimate alarm.

5.3 Interface to TLB

To access the underlying ITTC functionality, client programs link against the TLB code and call functions in its external interface. Since it is possible to perform all the standard RSA cryptographic operations (encrypt, decrypt, sign, verify) with an ITTC-style RSA key, existing applications can be rewritten to use these keys by replacing conventional RSA calls with calls to TLB functions.

This approach has some drawbacks, most notably that it requires effort to go through each application to find RSA cryptographic calls and replace them with their ITTC counterparts. Fortunately, the design of SSLeay allows us to implement the TLB interface more elegantly. In SSLeay, RSA keys (type `RSA *`) contain not only the data corresponding to the keys (N, e, d, etc.) but also pointers to functions that perform the four basic RSA operations. This form of polymorphism allows us to reduce the entire TLB interface to a single function call:

```
RSA *ITTC_load_RSA_key(char *ittckeyfile);
```

This function returns an RSA object whose "function table" points to functions that perform ITTC-style decryption and signing (encryption and signature verification are handled just like an ordinary RSA key). Modifying an application to handle ITTC-style keys is now a matter of changing only one piece of code, namely the code that loads RSA keys. Once the key is loaded in this manner, application code cannot distinguish an RSA object obtained through ITTC from a conventional RSA key, and no other code needs to be modified.

Section 6 contains some examples of applications that used this technique to support ITTC-style shared keys. This object-oriented method of encapsulating functionality with data can be carried over to other cryptographic architectures (e.g. Java's JCA or Microsoft CAPI) that support provider interfaces.

6 Example applications

6.1 Intrusion tolerant certification authority

The ITTC-based certificate authority application signs incoming certificate requests with the appropriate shared private key. The application itself is based on Eric Young's ca certificate authority program, which is included with SSLeay. As described in Section 5.3, adapting the existing code to handle ITTC keys involved only a minimal change to the code, as seen in Table 1.

```
#ifndef ITTC
  BIO_read_filename(in,keyfile);
  pkey=PEM_read_bio_PrivateKey(in,NULL,NULL);
#else
  pkey=EVP_PKEY_new();
  rsa=ITTC_load_RSA_key(keyfile);
  EVP_PKEY_assign_RSA(pkey,rsa);
#endif
```

Table 1: ITTC Certificate Authority Code Patch

The modified certificate authority (ittc_ca), accepts the same inputs and command line arguments as the original program. For example, the command

```
ittc_ca -config ca.cnf -in request.req
    -out newcert.pem -keyfile sharedkey.rsk
```

verifies the self-signed standard-form certificate request in request.req, signs it with the shared RSA key sharedkey.rsk, and writes the PEM-encoded X509 result in newcert.pem. The file ca.cnf is an internal SSLeay config file.

In most applications where an ITTC private key is used, it is necessary to obtain a signed certificate for the corresponding public key from a well-known issuer (e.g. when using ITTC to secure a Web server's private key, see Section 6.2). The ittc_req program generates correctly-formatted, self-signed certificate requests from ITTC-style shared keys. The command

```
ittc_req -config ca.cnf -new
      -key sharedkey.rsk -out request.req
```

compiles a standard-form certificate request for the shared key sharedkey.rsk and applies the shared key once to sign it. The output (request.req) is indistinguishable from a certificate request generated from a conventional RSA key and can be sent off to any certificate authority (e.g. Verisign) to be signed.

6.2 Distributed web server

In addition to securing the private key for a certificate authority, ITTC can protect private keys for a Web server. The technique of Section 5.3 was applied to the source code for ApacheSSL 1.2.6 to produce an ITTC-enabled secure server. Since Apache already used SSLeay to handle its public-key cryptography, the entire extent of our modifications to Apache consisted of the three lines of C code shown in Table 2.

```
#ifdef ITTC
   pConfig->prsaKey =
            ITTC_load_RSA_key(szPath);
#else
   pConfig->prsaKey=RSA_new();
   PEM_read_RSAPrivateKey(f,
               &pConfig->prsaKey, NULL);
#endif
```

Table 2: ApacheSSL Code Modification for ITTC

With this change in place, Apache accepts ITTC-style private keys and treats them as if they were conventional RSA keys. An administrator simply generates an ITTC shared key, generates a self-signed certificate request with the ittc_req application (see Section 6.1), and sends this request to a recognized certificate authority. Once the signed certificate is received, Apache is configured to use the certificate and corresponding key with the following entries in its configuration file, httpd.conf:

```
SSLCertificateFile
     /usr/local/httpd/SSLconf/cert.pem
SSLCertificateKeyFile
     /usr/local/httpd/SSLconf/privatekey.rsk
```

Once the Web server is started up, it will present the certificate cert.pem to Web browsers and use the ITTC shared key privatekey.rsk to decrypt and/or sign responses.

7 Performance

The following timing measurements were taken on six Intel-based PC's. Two were running Solaris 2.6 at 333Mhz and four were running Windows NT at 450Mhz. The machines were connected via a 10base-T Ethernet. Clearly, a faster local network would improve our performance numbers.

Table 3 shows the latency and throughput for several key-sizes and sharings. Latency is the time that a client has to wait for a request to be serviced. Throughput is the number of requests that can be serviced per second. The table also lists the number of concurrent threads running on the *client*. The more threads run on the client the higher the load on the servers. Load balancing causes each share server to see some fraction of the requests. For example, when 10 threads are used the client makes 10 concurrent requests. If a 3-out-of-5 sharing is in use, each share server will service an average of $\frac{3}{5}$ of the total requests, or an average of 6 concurrent requests.

If the requests are processed serially, throughput depends only on latency, but when multiple clients connect to the servers at once load-balancing increases throughput. Increasing the number of simultaneous connections slows down each individual request but increases the throughput. Unless stated otherwise, the number of simultaneous connections was set to two.

A larger key corresponds to more communication and longer computations, so as the number of bits in the key increases, the latency increases and the throughput decreases. In every configuration, the 1024-bit key was significantly faster than the 2048-bit key.

Latency is affected by both the number of servers involved in a request and the total number of servers. For a fixed threshold, a larger total number of servers makes load-balancing more effective by distributing concurrent requests over more servers. This keeps the servers from being inundated by requests. When the total number of server is fixed, a higher threshold is likely to increase latency. To see this recall that a request completes only once all the servers in the coalition respond. With a larger coalition it is more likely that a busy server is included in the coalition. Hence, the time until the request completes is likely to increase.

The ratio of servers used to the total number of servers is the dominating factor in throughput. For example, with the 3-out-of-5 sharing, this ratio would be 0.6. As this ratio shrinks, a smaller portion of the servers is required per request, and more simultaneous requests can be processed with the same performance.

Table 3 also compares shared keys versus standard non-shared RSA keys. As expected, latency is larger when using a shared key than when using a non-shared key. The use of the shared key requires communication with share servers which is not required when using a non-shared key. Likewise, throughput is lower when using a shared key, but throughput doesn't degrade as much as latency because of load-balancing and multi-threading of the share servers.

As the number of simultaneous requests (threads) grows, latency also grows, because of the higher load on the servers. However, throughput improves because of load balancing and because the servers can service some threads while other threads are idle.

Table 4 shows what happens when some servers are taken offline. This is implemented by putting the share-servers in "suspended" mode. In this mode, a server can accept connections, but immediately closes them. Performance is identical when a server is simply shut down.

The TLB detects all offline servers in the coalition in one pass. When too many servers are offline, TLB

	thr.	no sharing		2-out-of-3		2-out-of-4		3-out-of-4		3-out-of-5	
1024	2	0.067s	29.2/s	0.411s	4.86/s	0.403s	4.95/s	0.639s	3.08/s	0.638s	3.13/s
1024	10	0.070s	28.9/s	1.343s	7.14/s	1.180s	7.90/s	1.790s	4.83/s	1.755s	4.90/s
2048	2	0.370s	5.38/s	1.434s	1.38/s	1.268s	1.55/s	1.978s	1.01/s	1.975s	1.00/s

Table 3: Latency and throughput as a function of key size

	2-out-of-3		2-out-of-4		3-out-of-4		3-out-of-5	
0–offline	0.411s	4.86/s	0.403s	4.95/s	0.639s	3.08/s	0.638s	3.13/s
1–offline	0.604s	3.26/s	0.532s	3.72/s	0.979s	2.01/s	0.887s	2.19/s
2–offline			0.763s	2.60/s			0.967s	2.03/s

Table 4: The effect of offline servers (1024-bit key)

returns an error code. For example, the 2-out-of-3 sharing will return an error code if two servers are offline, since the remaining server cannot service a request by itself.

Taking servers offline increases latency because some requests have to be issued multiple times, with different coalitions. Also, as servers are taken offline, load-balancing forces the remaining servers to service more requests, further increasing latency. Throughput also suffers when servers are taken offline, for the same reasons as latency.

Performance drops more when going from 0–offline to 1–offline than it does when going from 1–offline to 2–offline. 2–offline is worse than 1–offline since in the worst case requests may need to be issued three times until a valid coalition is found. As servers are taken offline, the sharings that use a larger fraction of the total servers degrade the most. For example, taking one server offline affects the 2-out-of-3 and 3-out-of-4 sharings more than the 2-out-of-4 sharing. This is because load balancing is more effective when the number of servers per request is small compared to the total number of servers.

Table 5 shows the effects of corrupt servers. Corrupt servers act just like normal servers, but they return invalid responses. In the tests used to find this data, corrupt servers compute the correct response, then return twice that value. When an invalid response is detected all servers in the coalition are checked for corruption. Hence, all corrupt servers in the coalition are detected in one pass.

Latency grows dramatically when corrupt servers are detected. There are two causes for this growth. First, just as with offline servers, requests may need to be issued multiple times. Second, when a corruption is found, each server in the corrupt coalition must be checked to see if it is corrupt (see Section 4.1). Comparing this chart to the previous chart, it becomes apparent that a corrupt server adds approximately 2.7 times latency penalty as an offline server in the same configuration. As with offline servers, throughput suffers as servers are corrupted, for the same reasons as latency.

Table 6 shows the performance of a web-server and certification authority, both using shared and non-shared private keys of size 1024-bits and 2048-bits. The web-server was tested by repeatedly establishing an SSL connection and issuing a `HEAD` request to a URL on the web-server. The number of threads listed in the table reflects the number of concurrent threads used by the test program when establishing connections to the web server.

In the web-server, both latency and throughput are worse with a shared-key than with a non-shared key, but not by much. With 10 simultaneous connections, latency is only 24% higher for a shared key and throughput is only 17% worse. For most applications, this slowdown is insignificant considering that SSL session establishment is only a small part of the interaction with the web server.

8 Related work

Fray, Deswarte and Powel [8] and Deswarte, Blain and Fabre [5] describe an encrypted file system where file keys are distributed using Shamir secret sharing across several key servers. Keys are reconstructed every time a file is accessed. In contrast, by using threshold RSA, the ITTC system never reconstructs long term private keys in a single location.

	2-out-of-3		2-out-of-4		3-out-of-4		3-out-of-5	
0–corrupt	0.411s	4.86/s	0.403s	4.95/s	0.639s	3.08/s	0.638s	3.13/s
1–corrupt	0.955s	2.06/s	0.823s	2.40/s	1.478s	1.31/s	1.334s	1.50/s
2–corrupt			1.536s	1.28/s			2.084s	0.95/s

Table 5: The effect of corrupt servers (1024-bit key)

	key-size	thr.	no sharing		2-out-of-3		2-out-of-4		2-out-of-4 1–offline	
web-server	1024	2	0.209s	9.50/s	0.644s	3.10/s	0.515s	3.84/s	0.535s	3.73/s
web-server	1024	10	1.097s	8.25/s	1.362s	6.85/s	1.503s	6.03/s	1.543s	5.66/s
web-server	2048	2	0.389s	5.05/s	1.527s	1.31/s	1.484s	1.34/s	1.509s	1.32/s
CA	1024	2	0.067s	29.2/s	0.411s	4.86/s	0.403s	4.95/s	0.532s	3.72/s
CA	2048	2	0.370s	5.38/s	1.434s	1.38/s	1.268s	1.55/s	1.749s	1.12/s

Table 6: Usage of ITTC in a CA and web server

The Ω system [12], built at AT&T, also uses threshold cryptography to protect private keys. Ω supported a Certification Authority (CA) used at AT&T. It was the first system to demonstrate the practicality of threshold cryptography. We note that Ω does not support distributed key generation, detection of corrupt servers or the ability to refresh shares in case a share server is compromised.

The proactive security toolkit [2] built at IBM focuses on using proactive security applied to DSS to protect private DSS signing keys. The system shares a DSS key among a number of servers, and proactively refreshes these shares once every predetermined time period (e.g. once a day). Interestingly, DSS and RSA have different sharing properties. RSA keys are easy to share (so that a signature can be generated without reconstructing the key), but are hard to generate distributively (so that none of the participants know the private key). On the other hand, DSS keys are easy to generate distributively, but are harder to use for threshold signatures.

Our particular implementation of threshold RSA is based on one of several possible algorithms. We chose an algorithm for threshold RSA that is best suited to handle a small number of share servers (i.e. less than six). For a larger number of servers one could use a recent algorithm due to Shoup [15].

9 Conclusions

The ITTC system enables applications to store their private keys in an intrusion tolerant fashion. Penetrating a few share servers reveals no information about the private key. Penetrating a client is detected through our use of sequence numbers and does not expose any shared keys. Our system eliminates single points of failure by never reconstructing a shared private key in a single location. Even key generation is done in shared form. The system detects and corrects offline and corrupt servers. For instance, one of the share servers can be taken offline for maintenance without affecting system behavior.

ITTC is easy to embed into existing applications. We built a web server and a CA in which private keys are managed using ITTC. Our performance figures show that the cost of using ITTC is reasonable. This is especially true in the case of a web server where session key exchange is only a small fraction of the total work performed by the server. Session key exchange is not done too frequently since both the browser and server cache session keys.

Acknowledgments

Research supported by DARPA/ITO. We thank Bill Aiello and Sivaramakrishnan Rajagopalan (Raj) for their help throughout the project.

References

[1] N. Alon, Z. Galil, M. Yung, "Dynamic re-sharing verifiable secret sharing against a mobile adversary", in Proceedings of the 1995 European Symposium on Algorithms (ESA), pp. 523–537.

[2] B. Barak, A. Herzberg, D. Naor, E. Shai, "The proactive security toolkit and applications", to appear in the 6th ACM Conference on Computer and Communications Security, 1999.

[3] D. Boneh, M. Franklin, "Efficient generation of shared RSA keys", in Proceedings Crypto' 97, pp. 425–439.

[4] Y. Desmedt, G. Di Crescenzo, M. Burmester, "Multiplicative non-abelian sharing schemes and their application to threshold cryptography", Proceedings ASIACRYPT '94, pp. 21–32.

[5] Y. Deswarte, L. Blain, J Fabre, "Intrusion tolerance in distributed computing systems", Proceedings IEEE Symposium on Security and Privacy, Oakland, 1991, pp. 110–121.

[6] Y. Frankel, "A practical protocol for large group oriented networks", Eurocrypt 89, pp. 56–61.

[7] Y. Frankel, P. Gemmel, P. MacKenzie, M. Yung, "Optimal-resilience proactive public-key cryptosystems", Proceedings FOCS '97, pp. 384–393. .

[8] J. Fray, Y. Deswarte, D. Powell, "Intrusion tolerance using fine-grain fragmentation-scattering", Proceedings IEEE Symposium on Security and Privacy, Oakland, 1986, pp. 194–201.

[9] P. Gemmel, "An introduction to threshold cryptography", in CryptoBytes, a technical newsletter of RSA Laboratories, Vol. 2, No. 7, 1997.

[10] M. Malkin, T. Wu, D. Boneh, "Experimenting with shared RSA key generation", Proceedings of the Internet Society's 1999 Symposium on Network and Distributed System Security (SNDSS), pp. 43–56

[11] Public Key Cryptography Standards (PKCS), RSA Labs, available at `http://www.rsa.com/rsalabs/pubs/PKCS/`

[12] M. Reiter, M. Franklin, J. Lacy, R. Wright, "The Ω key management service", Proceedings of the 3rd ACM conference on Computer and Communication Security, 1996.

[13] T. Rabin, "A simplified approach to threshold and proactive RSA", Proceedings of Crypto' 98.

[14] A. Shamir, "How to share a secret", Comm. of the ACM, Vol. 22, 1979, pp. 612–613.

[15] V. Shoup, "Practical threshold signatures", to appear.

[16] E. Young, SSLeay, `http://www.ssleay.org/`

Brute Force Attack on UNIX Passwords with SIMD Computer

Gershon Kedem, Yuriko Ishihara
Computer Science Department
Duke University
Box 90129
Durham, NC 27708-0129

Abstract

As computer technology improves, the security of specific ciphers and one-way hash functions periodically must be reevaluated in light of new technological advances. In this paper we evaluate the security of the UNIX password scheme. We show that the UNIX password scheme is vulnerable to brute-force attack. Using PixelFlow, a SIMD parallel machine, we are able to "crack" a large fraction of passwords used in practice [12] in 2-3 days of computation. We explain how a SIMD machine built in today's technology could "crack" any UNIX password in two days. We also describe in this paper a simple modification to the UNIX password scheme that makes it harder to break encrypted passwords using dictionary and brute force attack, thus extending the useful life of the UNIX password scheme. The modified password scheme is compatible with the existing password scheme.

0. Introduction.

Single Instruction Multiple Data (SIMD) machines is a class of parallel machines that are made of a large number of simple processors all executing in unison the same instruction sequence, each processor computing on a different data set. It is relatively inexpensive to build SIMD machines with a very large number of processors. Since the processors are simple (typically 8-bit processors), processors use near neighbor connections, and processors do not use local instructions decoding, it is possible to integrate a large number of SIMD processors into a single IC. In current technology (0.2μ CMOS) it is possible to integrate 512-1024 processors each with 1-2 KB of RAM on a single IC. Moreover, since instruction decoding is external to the processors, it is easy to pipeline the processors and make them run at high frequency. In today's technology it is possible to build a SIMD processor array that execute one instruction per clock cycle at 1-GHz.

SIMD machines are very effective for performing regular computation on large data sets. Examples of applications that require large computing power and could effectively use SIMD machines are: real-time image analysis and real-time image generation. Both applications do relatively simple calculations on a large set of pixel data. Computation at each pixel is, for the most part, independent from the values of all but a few neighboring pixels.

SIMD machines are currently out of vogue since SIMD machines perform poorly on applications that require complex decisions and applications that have complex data dependencies. In either case the SIMD processor array could not be used effectively. In the first case, only a small fraction of the processors actually do useful work. In the second case, the machine spends a large portion of the time moving data between processors rather than doing useful work. SIMD machines are notorious for being hard to program. The SIMD machine programming-model is very different from the conventional programming model, and it is hard to port applications that run on conventional machines to SIMD machines.

However, SIMD machines are very effective for brute force cryptanalysis. First, most ciphers are very simple algorithms, made up of loops and straight-line code. Therefore, it is relatively straightforward to implement ciphers on SIMD machines, and the resulting programs harness the full power of the machine. Second, brute force cryptanalysis is "embarrassingly parallel". By assigning a different key to each processor it is possible to simultaneously check a large number of keys. The processors do not communicate, and all the processors execute most of the time. As a result, brute force cryptanalysis can achieve a system performance that is close to the theoretical performance-limit of the machine.

We have used the PixelFlow machine [2] to show that UNIX passwords are vulnerable to brute force cryptanalysis. The PixelFlow machine is an experimental graphics engine built at the Computer Science Department at UNC-Chapel Hill in cooperation with Hewlett Packard Corporation. The PixelFlow machine includes a large SIMD array of 8-bit pixel processors running at 100MHz. We programmed 147,456 PixelFlow SIMD processors to do brute force cryptanalysis of 40-bit RC4 cipher and the UNIX `crypt` password scheme.

In section 1. we describe the PixelFlow machine, and the SIMD array we used for the computation. Section 2. describes a brute-force attack using PixelFlow and its implications for UNIX security. We also describe the capabilities of a SIMD machine that one could build with today's technology and its security implications.

In section 3. we propose a way to modify the UNIX password scheme to make it resistant to brute force crypt-analysis.

1. The PixelFlow Machine.

PixelFlow is a heterogeneous parallel machine designed for a special purpose, high-speed and high-quality image generation. A PixelFlow system consists of at least one chassis, which includes up to nine Flow units. Each Flow unit has a SIMD array of 8,192 (8K) Processing Elements (PEs) running at 100MHz.

As shown in Figure-1, each Flow unit includes both a Geometry Processor (GP) board and a Rasterizer Board (RB). The GP board has two 180 MHz. PA-RISC-8000 CPUs, 128MB SDRAM memory, and a custom ASIC, the RHInO (Runway Host and I/O) that connects the processors with memory, the geometry network and the Rasterizer Board. The geometry network is a high-speed packet-routing network that connects the GPs to each other. In addition an I/O interface card connects the network to a host workstation.

The heart of the Rasterizer board is a SIMD (128x64) array of 8192 processing elements (PEs). The PE array is implemented on 32 logic-enhanced memory chips (EMCs), each containing 256 PEs. Figure-2 shows a block diagram of an EMC. Each PE is an eight-bit wide processor consisting of an arithmetic-logic unit (ALU), two registers and 384 bytes of local memory. This includes 256 bytes of main memory and four 32-byte partitions associated with two I/O ports. A linear expression evaluator computes values of the bilinear expression $A \cdot x+B \cdot y+C$ in parallel for every PE; the pair (x,y) is the address of each PA in the SIMD array. In addition, the Rasterizer board has a Texture/Video Subsystem that we did not use and will not describe here. For a full description of the PixelFlow system the reader should refer to [1, 2, 3].

Two Image Generation Controller ASICs (IGCs) control the rasterizer. The IGCs parse the instruction stream from the Geometry Processor board and issue micro-instructions to the SIMD PE array. A DMA unit transfers a stream of instruction from the GP SDRAM memory to IGC units and the SIMD array.

To program the SIMD array one loads a program into one of the GP CPUs. The GP processor, by executing the program, generates a sequence of microcode instructions for the SIMD array. The GP is used to control the SIMD array and to communicate with a host workstation. Since the system was designed for image generation, the SIMD array has a high bandwidth connection to the Texture/Video subsystem, but only a very limited (1-bit) connection back to the GP.

The programming environment for the SIMD array is limited. A set of C++ functions implements an assembly language instruction set for the SIMD array. A functional level simulator is used for program development and debugging. The SIMD-array instruction set was designed to efficiently execute image generation code. Instructions can operate on a single byte, two, four, or eight bytes of data. Most instructions take two or three clock cycles per byte to execute. The SIMD instruction set is quite conventional except the set of linear expression-evaluation instructions. The instructions are memory to memory instructions. The instruction set includes the usual integer and bit-wise logical

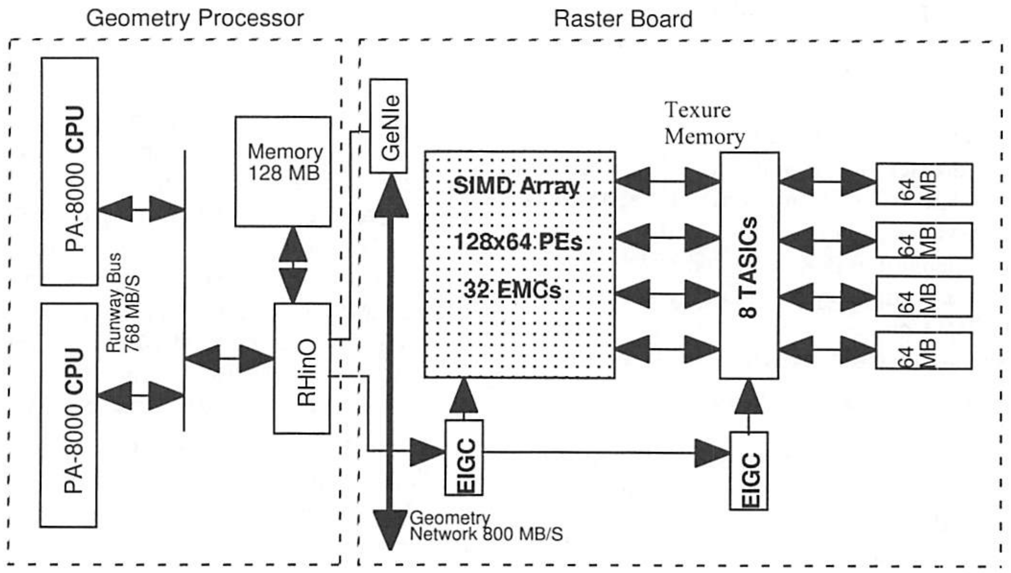


Figure-1: PixelFlow Unit, Block Diagram.

instructions. However, the PixelFlow SIMD array is missing indexing instructions. The PixelFlow hardware does not support the ability to take a memory value and use it as an index into memory. The PixelFlow machine was designed with image generation in mind, and the designers did not see the need for such capability. It turned out to be a major performance issue for implementing both the RC4 cipher and the UNIX `crypt` function (See section-2).

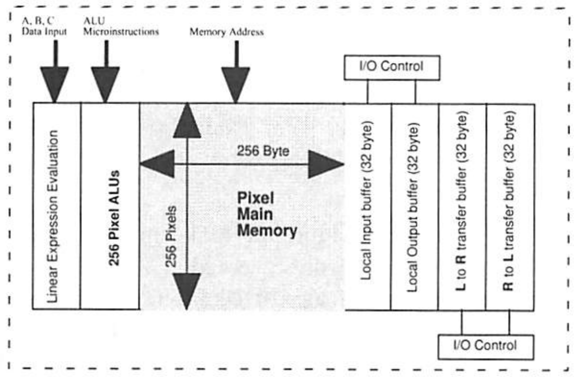

A, B, C Data Input
ALU Microinstructions
Memory Address
I/O Control
Linear Expression Evaluation
256 Pixel ALUs
256 Byte
256 Pixels
Pixel Main Memory
Local Input buffer (32 byte)
Local Output buffer (32 byte)
L to R transfer buffer (32 byte)
R to L transfer buffer (32 byte)
I/O Control

Figure-2: PixelFlow Enhanced Memory Chip (EMC), Block Diagram.

2. Brute Force Cryptanalysis with PixelFlow

Brute force cryptanalysis is the most straightforward cryptanalysis attack. The attacker uses raw computing power to find a key for the cipher by trying all possible keys. Typically the attacker has either a ciphertext-cleartext pair or ciphertext only. When the attacker is analyzing passwords, only the encrypted password is needed. Typically, brute force attack is carried either by using a collection of general-purpose computers or by using special-purpose hardware.

General-purpose machines provide a very flexible cryptanalysis platform. The machines can be programmed to attack many different ciphers. Nowadays when most computers are networked, it is not unusual to mount a brute force cryptanalysis attack using hundreds or thousands of computers. The attack is carried most often at off-hours, using the computers' "idle cycles". However, general-purpose computers are relatively slow. Using today's general-purpose computer cluster, it is practical to attack 40-bit keys, but 56-bit or 64-bit ciphers, in practice, are still out of reach.

Building special-purpose hardware for brute force cryptanalysis provides a very effective (but expensive) way to attack ciphers. Using today's technology (say, 0.2μ CMOS technology) one can build special-purpose machines that can "crack" 56-bit and 64-bit ciphers in a few seconds to a few days of computation. However, building special purpose hardware is time consuming and very expensive. A typical hardware development project could cost $10+ million and could last two to three years. At the end, all one has is a machine that can break a single cipher. If breaking the cipher is important enough, this approach makes sense; otherwise special-purpose hardware is too expensive for all but a few very large organizations or governments. Cryptanalysis attack using special-purpose hardware could be defeated by choosing another cipher or by making slight modifications to existing ciphers.

Building a programmable machine for cryptanalysis is an in-between approach. On one hand, the machine is programmable and could be used to analyze different ciphers; on the other hand, the machine could bring large computing power to the attack. Blaze et. al. [13] proposed using Field Programmable Gate Arrays (FPGAs) for mounting such an attack. Using FPGAs to implement ciphers in hardware could still require a large design effort. Designing circuits to implement ciphers at gate-level is a hard and error-prone task. Debugging the system is also hard and error-prone. However, once the design is complete, it could be mapped relatively quickly into FPGA arrays. FPGAs suffer from two problems: capacity and speed. FPGAs devote large portion of the silicon area to switches and wires. As a result, the number of gates one could actually use is much lower than the number of gates the technology supports. Second, since the on-chip gates are connected via switches, in practice the system clock speed is much lower than advertised. As a result, using FPGAs one could only built cryptanalysis systems with moderate performance.

We set out to evaluate the use of SIMD machines for brute force cryptanalysis. As was explained in the introduction, SIMD machines could provide cost effective way to mount brute force attacks. SIMD machines are fully programmable, and therefore it is quite easy to implement complex algorithms and search strategies. As noted before, when running brute force cryptanalysis code, the machine can approach its theoretical performance limit.

We used PixelFlow, an existing SIMD machine, in order to answer the following questions:

a. How hard is it to program ciphers and complex search strategies on a SIMD parallel machine?
b. What performance can one get out of the system when mounting realistic attacks?

c. What special features should SIMD systems have so they could support effective cryptanalysis attacks?
d. How powerful is the PixelFlow system? Can we compromise the security of ciphers used in practice?
e. If one is to design a new SIMD machine using current technology (0.2μ CMOS), a machine designed specifically for brute force crypt-analysis, how powerful a system one can build and at what cost?

We programmed two ciphers for the PixelFlow machines: RC4 [11] and the UNIX `crypt` algorithm [5, 6, 8, 9, 11]. The programming was done in machine assembler. Learning how to use the tools took longer than we expected, but the programming task proved to be no harder than writing assembler for any machine. The only difficulty we ran into was the fact that PixelFlow does not have index registers and it does not gracefully support table lookups. This turned out to be a major performance issue. On a "normal" machine array-lookup takes one or two instructions. On PixelFlow it takes time proportional to the array size. Using some clever programming, we were able to reduce it to time proportional to the square root of the array size [4]. Since the RC4 and DES algorithms both are dominated by table lookups, the performance we were able to achieve fell far short of the performance we expected. We estimate that a PixelFlow machine that supports indexing would run the RC4 code ~32X faster and run DES ~16X faster.

We used a PixelFlow system with two full chassis for a total of 18 SIMD arrays. Each array has 8K (8192) PEs, so the total number of PEs we used is 147456. At any given iteration the machine checks 147456 different keys in parallel. We used the linear expression evaluation unit to distribute a unique key to each PE. All PEs run in unison, and at the end of each iteration, the PEs check to see if any of the keys produced a match. The RC4 algorithm was timed at approximately 1000 key checks every 3.725 seconds per each PE. Since we have 147456 PEs, the system checks 38,804,210 keys per second. This enables us to check all 40-bit combinations in ~28,335 seconds (~ 7.87 hours) and on the average find a key in less than four hours.

2.1 Checking UNIX Passwords.

Checking UNIX passwords proved a bit more complex. To be able to check passwords rather than DES keys we devised a simple algorithm that translates a number into a unique password in a given class. Assume that we want to check all possible passwords with lower case letters. We construct a character array with 26 characters with the lower case letters. Translating the number *I* into the password *P* is just a base translation from binary to base 26, using the characters as the base digits. This assigns a unique password to each number. Simple modifications to the base translation algorithm let us check different classes of passwords. For example, checking all passwords with lower case letters and one digit in the third position is done with a base-change to base-26 in all positions except a base-10 change in the third position. Similar modifications let us check many different password classes. The idea is to check password classes that are most commonly used in practice. Eugene H. Spafford reports [12] that 28.9% of all passwords observed in his study had lower case letters only, and 38.1% had a mixture of lower and upper case letters. Brute-force attacks on these two password-classes alone make it very likely that a password would be compromised in a short time. For example, Table-1 shows that all lower case only passwords could be checked in 3.19 hours.

The UNIX `crypt` algorithm took approximately 6 seconds per 1000 passwords checked per PE. This yield a system level performance of 24,576,000 UNIX passwords checked per second (or 614,400,000 DES encryptions per second). Table-1 shows the time it would take to check all passwords in a given class. In the table we used LC for lower case letters, UC for upper case letters, D for digits, and P for all other characters.

2.3 Building SIMD machine in today's technology

The PixelFlow machine was built in the mid-90's using 0.5μ CMOS technology with three layers of metal. Today's CMOS technology has 0.18μ-0.2μ features, and the technology supports up to six layers of metal. Linear shrink alone yields a factor of ~6x improvement in density and a factor of ~4x improvement in speed. The PixelFlow machine was designed for high-speed image generation, and the SIMD array is only a small part of the overall system. Moreover, the SIMD array was not optimized for performance since it is already more powerful than other system components. The SIMD array is never the limiting factor in image generation, and rarely if ever is it used to its full potential. If one is to design a SIMD machine for brute force cryptanalysis, one could do better than just scale up the PixelFlow design.

We have constructed a "paper design" of a SIMD machine optimized for brute force cryptanalysis, using what we have learned from programming PixelFlow. Most of the design decisions for PixelFlow were carried over to our new design. The system is a heterogeneous machine made up of many SIMD arrays. Each SIMD array is controlled by a general purpose (GP) computer that also serves as a communication controller. The

SIMD arrays are connected to an ethernet network and are controlled by a general-purpose workstation. Like PixelFlow the GP processor has large SDRAM memory. The GPs are used to generate instruction streams for the SIMD array, and a DMA controller is used to load the instruction into the array.

The processing elements (PEs) are upgraded to improve on the PixelFlow design. Like in PixelFlow, processors are 8-bit wide. Each processor has 1K bytes of local memory and a set of 32 registers. The processors use a four stage pipeline (register fetch, execute, memory access, register store). Pipelining should improve system speed by 3x-4x. We (conservatively) estimate that 512 processors could be integrated into a single IC. We estimate the clock speed to be 1 GHz. Each SIMD array has 64 processors' ICs (32K PEs) with its own GP, SDRAM buffer and ethernet connection. Each SIMD array is implemented on a single printed circuit board. We estimate the replication cost of each SIMD array board to be $3,000. For a replication cost of about $100,000 one could build a system with 1,048,576 (1Meg) PEs. We estimate that such a system will deliver at least 1000X better performance than the PixelFlow system we used. Table-1 also lists the estimated time it would take to check different password combinations on the new machine.

Class	Combinations	PxFl hours*	New Design hours*
LC only	2.82E+11	3.19	0.003
LC + 1-UC	2.18E+12	24.59	0.025
LC + 2-UC	1.47E+13	165.77	0.166
LC + 1-D	8.37E+11	9.46	0.009
LC + D	3.51E+12	39.70	0.040
LC + UC	6.23E+13	703.71	0.704
LC + UC + 1-D	9.40E+13	1062.20	1.062
LC + UC + D	2.18E+14	2467.86	2.468
All passwords	5.13E+15	58008.14	58.008
All DES keys†	7.21E+16	32578.12	32.578

Table-1: Times for checking different password classes. (*Estimated, †DES only)

3. Improving Password security.

Our brute force attack experiment, using PixelFlow's SIMD processor arrays, demonstrates that UNIX passwords are vulnerable to such an attack. Moreover, we argue that it is possible to build a SIMD machine that could check all possible UNIX passwords in about two days. Some UNIX installations require that users use "safer" passwords. They instrument the UNIX `passwd` program to force users' passwords to have characters from at least two or three of the four categories: upper-case letters, lower-case letters, digits, and other-characters. This modification to the UNIX `passwd` program only marginally improves performance against brute force attack, since the attacker can use that knowledge to direct her/his attack. For example: there are 1.67E+12 passwords that have one upper-case letter and the rest are lower-case letters. The "more complex" class of passwords that have exactly one upper-case letter, one digit and the rest lower case letters has 4.50E+12 passwords, only a factor of 4 more passwords. This factor is not large enough to make a difference. We also argue that it is possible to build machines that are able to "crack" any UNIX password in a day. Therefore we are recommending that the UNIX community adopt a modification to the existing password scheme. This modification is backward compatible; it is tunable to the hardware "state of the art," making it possible to make the password scheme more secure when common computers become faster, while keeping the login time to about a second.

The main idea is to add random bits to the password encryption, similar to the "salt" bits that are currently used to protect passwords against dictionary attacks. We call the new bits: "pepper" bits. Unlike the "salt" bits that are saved with the encrypted password, the pepper bits are used to encrypt the password, but are never saved.

Let say that we are using k pepper bits for password encryption. The k-bit vector P is repeated as many times as necessary to form a 64-bit word V ($V=<P, P, \cdots>$). Let $E(\boldsymbol{e}, \boldsymbol{s}, \mathbf{0})$ stands for the normal encrypted password, where $\boldsymbol{e}$ are the password bits, the $\boldsymbol{s}$ are the salt bits, and $\mathbf{0}$ is a 64-bit zero constant that is encrypted to get an encrypted UNIX password. Then the new encrypted password is: V+E($\boldsymbol{e}$, $\boldsymbol{s}$, V), where + stands for bit-wise exclusive-or. That is, the `crypt` algorithm is applied to the 64-bit word V (rather than to $\mathbf{0}$) and the result is XORed with V. Note that if the pepper bits are all zero, V is zero and the new scheme is identical to the current scheme. When a user selects a new password the system generates a random k-bit vector P. The system generates the vector V by repeating P as many times as necessary, and it uses <e, s, V> to generate the encrypted password. The system saves the encrypted password together with the salt bits, but it *does not save* the vector V. When a user logs in, the system checks the password supplied by the user as before. The system runs `crypt` 2^k times with all the possible values for V, computing V+E(e, s, V). If any of the 2^k possible values match the encrypted password, the login succeeds. Otherwise the login fails.

What we propose here is essentially a "whitening" technique [11]. Our proposed UNIX password scheme is similar to a proposal made by Udi Manber [7], but it improves on Manber's scheme. Unlike the scheme proposed by Manber, if one chooses k carefully, the probability of hitting a false positive <password, pepper> combination is smaller than the one proposed by Manber. The probability that V_1+E($e1$, s, V_1) = V_2+E($e2$, s, V_2) when $V_1 \neq V_2$ and $e1 \neq e2$ is $2^{-(64-k)}$ since the encryption function is a strong one-way hash function, both as a function of the ciphertext and as a function of the keys. If one chooses k=11, the probability of hitting a false positive combination is 2^{-53}, about the same probability as guessing a random password. Our scheme protects passwords against dictionary and brute force attacks by forcing the adversary to do 2048X more work, while keeping the probability of "false positive" guesses to the same probability of guessing a random password. Our experiments show that on current machines, adding 11 pepper bits keeps login time to about a second.

4. Conclusions:

In this paper we study the use of SIMD machines for brute force cryptanalysis. We show that SIMD parallel machines are very effective. We demonstrate that an existing machine, PixelFlow, can break many UNIX passwords that are used in practice. We argue that it is possible and practical today to build a machine that could "crack" any UNIX password in a day or two. Moreover, this machine could be programmed to break ***any 56-bit cipher*** in two days. We proposed a simple modification to the UNIX password scheme that makes the scheme 2048X more resistant to dictionary and brute force attack. We argue that as time goes by and computer hardware becomes faster and less expensive, reusable password schemes are becoming more vulnerable. The source of vulnerability is the fact that people must remember the password and should not keep a written copy of the password or store the passwords electronically. In practice, the set of passwords that people can remember is too small to offer strong protection against adversaries with large computing resources. The community will be well served by introducing new authentication schemes.

Acknowledgements:

PixelFlow was designed and built by a research team at the Department of Computer Science at UNC-Chapel Hill and the Hewlett-Packard Corporation. We are grateful to the UNC team for helping us use PixelFlow. Special thanks are due to Anselmo Lastra and John Eyles for their help.

References:

[1] Lastra, Anselmo, Steven Molnar, Marc Olano, and Yulan Wang, "Real-Time Programmable Shading", *Proc. of the1995 Symposium on Interactive 3D Graphics, ACM Siggraph*, April 1995.

[2] Eyles, John, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, and Nick England, "PixelFlow: The Realization", *Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware*, Los Angeles, CA, August 3-4, 1997, 57-68.

[3] Eyles, J. and Molnar, S., "PixelFlow™ Raterizer, Functional Description". UNC CS Department internal document, 1997.

[4] Eyles, John, "Privet Communication". 1998

[5] Feldmeier D.C., and P. R. Karen, "UNIX password security – ten years later", *Proceedings, UNIX Security Workshop,* August 1989.

[6] Simon Garfinkel and Gene Spafford, *Practical Unix Security*, O'Reilly & Associates, Inc., Sebastapol, CA, 1991.

[7] U. Manber, "A Simple Scheme to Make Passwords Based on One-Way Functions Much Harder to Crack," Computers & Security 15 (2) (1996), pp. 171-176.

[8] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1996.

[9] Robert Morris and Ken Thompson, "Password Security: a case history". In *UNIX Programmer's Supplementary Documentation, AT&T*, November 1979.

[10] Poulton, J., J. Eyles, and S. Molnar, "Breaking the Frame-Buffer Bottleneck with Logic-Enhanced Memories," *IEEE Computer Graphics and Applications*, November 1992, pp. 65-74.

[11] Bruce Schneier, *Applied Cryptography,* 2nd Edition, John Wiley & Sons, Inc. New York, 1996.

[12] Eugene H. Spafford, "Observing Reusable Password Choices", *In Proceedings of the 3rd Security Symposium*, Usenix, September 1992.

[13] M. Blaze, W. Diffie, R.L. Rivest, B. Scheier, T. Shimomura, E. Thompson, M. Weiner, "Minimal Key Length for Symmetric Ciphers to Provide Adequate Commercial Security", *A Report by an Ad Hoc Group of Cryptographers and Computer Scientists.* URL: http://theory.lcs.mit.edu/~rivet/bsa-final-report.ps

Antigone: A Flexible Framework for Secure Group Communication

Patrick McDaniel[*] Atul Prakash
EECS Dept.
University of Michigan
Ann Arbor
aprakash@eecs.umich.edu

Peter Honeyman
Center for Information Technology Integration
University of Michigan
Ann Arbor
honey@citi.umich.edu

Abstract

Many emerging applications on the Internet requiring group communication have varying security requirements. Significant strides have been made in achieving strong semantics and security guarantees within group environments. However, in existing solutions, the scope of available security policies is often limited. This paper presents Antigone, a framework that provides a suite of mechanisms from which flexible application security policies may be implemented. With Antigone, developers may choose a policy that best addresses their security and performance requirements. We describe the Antigone's mechanisms, consisting of a set of micro-protocols, and show how different security policies can be implemented using those mechanisms. We also present a performance study illustrating the security/performance tradeoffs that can be made using Antigone.

1 Introduction

IP multicast [Dee89] services are becoming more widely available on the Internet. The use of group communication based on IP multicast as a fundamental building block for a variety of applications such as video conferencing will increase. Many interesting applications that require group communication, such as video conferencing, collaborative applications, data casting, and virtual communities, are emerging on the Internet. These applications, depending on the perceived risks and performance requirements, require different levels of security. In this paper, we present a system, called Antigone, that provides mechanisms for building a range of security policies for group communication.

Existing approaches provide two predominant policies. Secure multicast systems [HM97, KA98] view groups as collections of participants that require a shared secret, called a *session key*, to secure application traffic. Group membership may change without affecting the security context. A potential security risk is that past members of the group may have access to current content. The advantage of this approach is low cost; rekeying after membership changes is not needed.

Conversely, secure group communication systems [Rei94] typically have threat models that require the changing of the security context after each membership change. Thus, protection from members not currently in the group is achieved at the cost of the context change. Some systems support stronger threat models than are strictly required by applications at a high performance cost.

The Antigone framework provides a flexible interface for the definition and implementation of a wide range of secure group policies. A central element of the Antigone architecture is a set of *mechanisms* that provide the basic services needed for secure groups. Policies are implemented by the composition and configuration of these mechanisms. Thus, Antigone does not dictate the available security policies to an application, but provides high-level mechanisms for implementing them.

To ease the task of application development, we also provide a facility for configuring rapidly a security policy from a set of standard policies that have been implemented using the Antigone mechanisms. An application is free to use these standard policies, if they are satisfactory for its purpose, or build its own using the high-level mechanisms provided by Antigone.

[*]This research is supported in part by the NASA Graduate Student Researchers Program, Kennedy Space Center, Grant Number 10-52613

Antigone is targeted for systems likely to require IP-multicast services. The majority of existing multicast based systems distribute high bandwidth content to a highly dynamic membership. A natural requirement of these systems is for efficient group and security management.

We see videoconferencing as representative of the types of applications that may benefit from the Antigone framework. However, we do not limit Antigone to continuous media systems. A fully functional secure group communication system may be built on the Antigone mechanisms.

While there are well known techniques for providing secure groups, Antigone has several goals that make it unique. We state the following as the five primary goals of Antigone.

1. **Flexible Security Policy** - An application should be able to use a wide range of security policies, with appropriate performance tradeoffs.
2. **Flexible Threat Model** - The system should support threat models appropriate for a wide range of applications.
3. **Security Infrastructure Independence** - Our solution should not be dependent on the availability of a specific security infrastructure.
4. **Transport Layer Independence** - The solution should not depend on the availability of any single transport mechanism (such as IP Multicast [Dee89]). On the other hand, our solution should be able to take advantage of IP-multicasts when they are available.
5. **Performance** - The performance overheads of implementing security policies should be kept low.

An early version of Antigone has been integrated into the `vic` [Net96] videoconferencing system. The result of that effort, called the Secure Distributed Virtual Conferencing (SDVC) application, was used to broadcast securely the September 1998 Internet 2 Member Meeting to several sites across the United States. The broadcast and other WAN tests indicate the viability of our approach. Using high speed cryptographic algorithms we are able to attain television-like secure video frame rates (30 frames/second) over a LAN. Details of the implementation and our experiences deploying SDVC can be found in [MHP98, AAC+99].

The remainder of the paper is organized as follows. Section 2 maps the design space of secure group communication policies. Section 3 outlines existing designs and techniques of secure group communication. Section 4 outlines Antigone's high-level, layered architecture. Section 5 presents the mechanisms that can be used to implement a wide range of group security policies. Section 6 shows that a wide range of security policies can be implemented using these mechanisms. Section 7 presents the available multicast transport modes of Antigone. Section 8 presents preliminary performance results for some of the policies that can be implemented using Antigone. Section 9 summarizes our conclusions and presents directions for future work.

2 Group Security Policies

Different group communication applications require different security policies, depending on their threat model and performance requirements. In this section, we point out some of the important dimensions along which group security policies vary. To address the requirements of a wide range of applications, Antigone attempts to provide a basic set of high-level mechanisms that can be used to implement a range of group security policies. The dimensions that are discussed below are: *session rekeying policy*, *application message policy*, *membership awareness policy*, and *failure policy*. The session rekeying policy defines the reaction of the group to changes in membership in terms of rekeying sessions. The application message policy defines the security guarantees with which application messages are transmitted. The membership policy dictates the availability of membership information to members. The failure policy defines the type of failures handled by the system. The remainder of this section describes the nature and implications of these policy decisions.

Antigone assumes that all group members are trusted, i.e., members do not actively attempt to circumvent the security of the system. Non-members however may attempt to intercept messages, modify messages, or prevent messages from being delivered.

2.1 Session Rekeying Policy

A common strategy to support secure group communication among trusted members is to use a common symmetric session key. An important policy issue for a group communication application is deciding when a session must be rekeyed, i.e., the old session key is discarded and a new session key is sent to all the members.

There is a close relationship between session rekeying and group membership. Applications often need protection from members not in the current *view*.[1] Therefore, changes in group membership require the session to be rekeyed. If rekeying is not performed after each change in membership, the view will not reflect a secure group, but indicates only which members are actively participating in the session. Past members may retain the session key and continue to receive content. Future members may record and later decode current and past content. In applications that do not need protection from past or future members, rekeying after membership events is unnecessary.

We define the *view group* to be the set of members in the current group membership view. A *key group* is the set of members who have access to the current session key. The key group may be a subset, a superset, or overlap the view group.

We say that a group security policy is said to be *sensitive* to an event if it changes the security context in response to the reception of the event. Typically, the security context is changed by distributing a new session key (rekeying).

Group security policy is often sensitive to *group membership events*. Group membership events include (1) `JOIN` event, which is triggered when a member is allowed to join the group; (2) `LEAVE` events, which is triggered when a member leaves the group; (3) `PROCESS FAILURE` events, when a member is assumed to have failed in some manner; and (4) `MEMBER EJECT` events, when a previously admitted member is purged from the group according to some group policy. In addition, sessions may be rekeyed when a specified time interval has passed since the last session rekey.

The sensitivity of a policy directly defines its threat model. For example, consider a group model that is sensitive to only `MEMBER EJECT` events. Because the session is rekeyed after member ejection, the key group will never contain an ejected member. Thus, the application is assured that no ejected member will ever have access to future content. However, the session content is not protected from processes that have left voluntarily, are assumed to have failed, or join in the future.

The sensitivity mechanisms in Antigone can be used to build a large number of session rekeying policies. In the following text, we define and illustrate four general purpose policies that are representative of the policies implemented in existing systems.

[1]A group *view* is the set of identities associated with the members of the group during a period where no changes in membership occur. If the membership changes (a member joins or leaves the group), then a new view is created. This is a similar concept to Birmans's group view [Bir93].

Time-sensitive Rekeying Policy
Groups implementing a time-sensitive policy periodically rekey, independent of group membership events. Periodic rekeying prevents the session key from "wearing out". The group attempts to guard against cryptanalysis by using a session key only for a limited period.

By periodically rekeying, the group may be protected from an adversary who wishes to block the delivery of new session keys. An adversary blocking rekeying messages may intend for the group to continue to use the current session key. With a time-sensitive rekeying policy, if a new key is not successfully established after the current session key expires, group members can choose to no longer communicate rather than use the expired session key.

Time-sensitive rekeying can be useful in a secure online subscription service. Paying members would periodically be sent a new key that is valid until the next subscription interval. The GKMP [HM97] protocol implements a time-sensitive rekeying policy.

Typically, systems implementing time-sensitive groups use *Key Encrypting Keys* (KEK) [HM97] to reduce the costs of rekeying. In systems that use KEKs, the key group contains all previous and current members of the group. This approach is limited in that any member may continue to access the group content after leaving. Systems that use KEKs cannot forcibly eject members without additional infrastructure.

All rekeying in Antigone is *session key independent*. Session key independence states that knowledge of one session key provides no information about others. With this independence, Antigone provides a stronger guarantee than time sensitive groups that use KEKs; a member who has left the group may continue to access the group content only until the next rekey. A past member cannot access current or future group content without again joining as a member.

Leave-Sensitive Rekeying Policy
Groups implementing a leave-sensitive policy rekey after `LEAVE`, `PROCESS FAILURE`, and `MEMBER EJECT` events. Thus, the key group may be arbitrarily larger than the view group.

The threat model implied by leave sensitive groups states that any member who has left the group will not have access to current or future content. For example, a business conferencing system that supports negotiations between a company's representatives and a supplier may benefit from leave-sensitive rekeying. Once the supplier leaves, a leave-sensitive rekey policy would prevent subsequent discussions from being available to the supplier, even if the supplier is able to intercept all the messages. The Iolus [Mit97] implements a leave-sensitive rekeying policy.

Join-sensitive Rekeying Policy
Groups implementing a join-sensitive policy rekey only after membership JOIN events. The threat model implied by join sensitive groups states that any member joining the group should not have access to past content. A join-sensitive rekeying policy, by itself, does not ensure that members who left the group or were ejected from the group are not able to access current session content. The assumption is that past members can be trusted to not be interested in current content.

In practice, a join-sensitive rekeying policy is likely to be used in conjunction with a time-sensitive or a leave-sensitive rekeying policy to limit the duration over which past members can access current session content.

Membership-sensitive Rekeying Policy
Groups implementing a membership-sensitive policy rekey after every membership event. The threat model implied by membership sensitive groups states that any group member joining the group will not have access to past content, and that members who have left the group will not have access to current or future content. This policy is the combination of leave-sensitive and join-sensitive rekeying.

Applications with comprehensive security requirements will likely need a membership-sensitive rekeying policy. Providing strong guarantees for message delivery (e.g. atomicity, reliability) often requires tight controls over the message content. Without tight controls, the guarantees may be circumvented. The RAMPART [Rei94] system provides a type of membership-sensitive service.

Other Rekeying Policies
Applications may require a combination of the above policies or may use different policies depending on the application state and the specific attributes of an event.

In the business conferencing application, for example, the policy may be to rekey only when a member with the role Supplier leaves, but not when a member with the role Representative leaves. Allowing policies that make distinction between members may allow the application to optimize rekeying.

In certain circumstances it may be important for the group to be more sensitive at certain times, but less at others. Groups may wish sensitivity to be a function of group size or resource availability. In this way, a group can provide the strongest security model that is supported by the network and host infrastructure.

Time-sensitive rekeying can be useful in combination with any of the other schemes. Time-sensitive rekeying in combination with membership-sensitive rekeying, for example, helps ensure that an adversary cannot prevent session rekeying indefinitely without detection after a membership change event.

2.2 Application Message Policy

An application message policy states the types of security guarantees required for application messages. For example, an application wishing to ensure that no party external to the group is able to access content will define confidentiality as part of its policy.

The most common types of message security guarantees are: *integrity*, *confidentiality*, *group authenticity*, and *sender authenticity*. Confidentiality guarantees that no member outside the group may gain access to the session traffic. Integrity guarantees that any modification of an application message is detectable by receivers. A session key is typically used to obtain these guarantees. However, some guarantees such as sender authenticity are difficult to obtain without additional security infrastructure. Antigone supports these four message guarantees.

Note that a single policy need not apply to every message. In most applications, different messages will require different guarantees, depending on the nature of the message and the assumed threat model.

2.3 Membership Policy

Identification of the membership within a group session is an important requirement for a large class of applications. As evidenced by a number of group communication systems, achieving strong guarantees for the availability and correctness of group membership can be costly. In providing availability, any change in membership requires the distribution of the new group membership *view* to each member.

Conversely, members in another class of systems need not be aware of group membership at all. This is the model used in typical multicast applications. In this environment, providing other services (such as reliability, fault-tolerance, etc.) is commonly left to the application.

A membership policy indicates the availability of group membership information. If the policy states that group membership be supported, the group *view* (membership information) is distributed to each member as needed.

Antigone provides mechanisms to distribute keys with and without membership information, primarily to allow higher-performance applications when members do not need membership information.

Currently, Antigone does not attempt to guarantee the confidentiality of group membership. In general, hiding the group membership from members and non-members is difficult to do in current networks without introducing noise network traffic. This is primarily because the ability to intercept messages on the network allows access to the source and destination of packets (in case of unicasts) and at the multicast tree (in case of IP multicasts). In mounting this *traffic analysis attack*, an adversary may deduce a close approximation of group membership.

2.4 Process Failure Policy

A process failure policy states the set of failures to be detected and the security to be applied to the failure detection mechanism. The defining characteristic of a failure detection mechanism is the definition of its fault model. The fault model defines the types of behavior exhibited by a faulty process that the mechanism will detect. Typical crash models include fail-stop, message omissions, or timing errors [Mul93]. In the strongest (Byzantine failure) model, a faulty process may exhibit any behavior whatsoever.

A process failure policy may also state the need for secure failure detection. In securing the failure detection, the group may be protected from the masking of process failures by an adversary. However, protecting the group from an adversary who attempts to generate false failures may be more difficult. Failures may be forced by blocking all communication between the group participants. This is a denial of service attack, which is difficult to address in software.

Antigone supports detection of fail-stop faults of group members. To prevent problems due to timing errors, synchronized clocks and timestamps are not used in the Antigone protocols. However, some mechanisms for failure detection and time-sensitive rekeying in Antigone do rely on timeouts at individual processes. A process whose clock progresses at an incorrect rate may take longer to detect failures of other processes (if its clock progresses too slow) or may mistakenly assume that there is a failure (if its clock progresses too fast and thus times out).

3 Related Work

In this section, we review several of the key concepts and known techniques in the design of frameworks for group communication. Two fields applicable to our investigation are secure group communication and design approaches for configurable protocols. We present some of the major findings in each of these areas.

Much of the existing technology on which group communication is based was originally implemented in the ISIS [Bir93] and later HORUS [RBM96] group communication systems. Using these frameworks, developers can experiment with a number of protocol features. One important feature of the HORUS system is the introduction of a comprehensive security architecture. An element of this architecture is a highly fault-tolerant key distribution scheme. Process group semantics are used to facilitate secure communication. The session key distributed to members in HORUS is maintained for the entire session, thus providing no sensitivity to membership change events. However, the vulnerability to attacks from past or future members is limited. Application messages have sender authenticity and may be confidential.

Virtual networks provide developers with an abstraction for building applications designed for (logically) local network traffic, but executed across across physically larger networks. The Enclaves system [Gon96] extends this model to secure group communication. Enclaves provides a leave-sensitive group, distributing a new *group key* after each member leave. Also, it is implied that the group key should be changed periodically (time-sensitive). Enclaves supports membership information but is not dependent on it. Messages have confidentiality and through point-to-point communication, sender authenticity.

The RAMPART system [Rei94] provides secure communication in the presence of actively malicious processes and Byzantine failures. The system uses secure channels between two members of the protocol to provide authenticity. Protocols depend greatly on the *consensus* of processes to reach agreement on the course of action. A membership-sensitive group is built on a secure group membership protocol. The security context is not changed through shared session keys, but the secure distribution of new group views. Messages have sender-authentication and integrity.

A limitation of many secure group communication systems is that they do not scale to larger networks. In [Mit97], Mittra defines the *1 effects n* failure, where a single membership change event effects the entire group. The Iolus system [Mit97], attempts to address this limitation by introducing locally maintained subgroups. Each subgroup maintains its own session key, which is replaced after a membership change event in the subgroup. Therefore, the effect of a membership change is localized to the subgroup. Iolus provides a leave-sensitive group, and all application messages are encrypted with the group session key. No membership information is distributed in Iolus.

Key hierarchies [WHA98, WGL98] provide an efficient alternative to subgrouping in achieving scalable, secure groups. A key hierarchy is singly rooted *n-ary* tree of cryptographic keys. The interior node keys are assigned by a session leader, and each leaf node key is a secret key shared shared between the session leader and a single member. Once the group has been established, each member knows all the keys between their leaf node key and the root. As changes in membership occur, rekeying is performed by replacing only those keys known (required) by the leaving (joining) member. Rekeying without membership changes (as needed in time sensitive groups), can be achieved by replacing the root key. Thus, the total cost of rekeying in key hierarchies scales logarithmically.

The Group Key Management Protocol (GKMP) [HM97] attempts to minimize the costs associated with session key distribution by loosening the requirement that each session key be independent of others. After being accepted into the group, newly joined members receive a *Key Encrypting Key* (KEK) under which a future session key will be delivered. A limitation of this group is that misbehaving members can only be ejected by the establishment of a new group. GKMP reduces the costs of authentication by introducing a peer-to-peer review process in which potential members are authenticated by different members of the group. The joining member's authenticity is asserted by existing members. GKMP provides a time-sensitive group by rekeying at the end of a session key's *lifetime*. Note that this is a key management protocol, and as such does not mandate how the session key is used. No membership information is provided to members.

The IPSec [KA98] standards attempt to achieve Internet security by introducing security mechanisms at the network layer. The Scalable Multicast Key Distribution (SMKD) [Bal96] standard extends this approach to the multicast environment. Intended as an extension to Core Based multicast routing [BFC93], SMKD uses the router infrastructure to distribute session keys. As the multicast tree is constructed, leaf routers obtain the ability to authenticate and deliver session keys to joining members. Thus, the cost of authentication and session key delivery can be distributed among the leaf routers. Similar to GKMP, SMKD provides a time-sensitive group which is rekeyed at the end of a session key's lifetime.

With respect to security, the policies implemented by these systems are essentially fixed. Applications must select a solution that provides a policy that best fits a minimal set of requirements. However, the accepted solution may provide unnecessary functionality at an increased cost.

The *micro-protocol* [HP94, HS98] design methodology is useful when constructing systems with dynamic protocol stacks. Using micro-protocols, a system designer may decompose the facilities provided by protocols into their atomic components. *Composite protocols* are constructed from a collection of the smaller micro-protocols. Differing facilities and guarantees may be provided through the composition of the micro-protocols. In [HS98], the authors define and demonstrate a suite of micro-protocols for maintaining group membership within a distributed application. Antigone uses the micro-protocol approach to achieve flexibility in implementing various security policies.

4 Architecture

Described in Fig. 1, the Antigone architecture consists of three software layers; the broadcast transport layer, the mechanism layer, and the predefined policy layer.

Though not typically associated with secure communication services, the broadcast transport component provides an abstraction for unreliable group communication. A reality of today's Internet is that there are varying levels of support for multicast services. Although significant effort has been expended on the development of WAN multicasting, no single solution has been found to meet the requirements of all users. Developers specify the level of multicast support of the target environment at run time. We describe the broadcast transport layer in Section 7.

The mechanism layer provides a set of mechanisms used to implement application policies. The mechanisms represent the basic features required for a secure group. Policies are flexibly defined and implemented through the selection and interaction of mechanisms. Associated with each mechanism is a micro-protocol [HS98]. The group protocol, called a *composite protocol*, is defined by the composition of the mechanism micro-protocols. We identify and describe the design of the Antigone mechanisms in Section 5.

The predefined policy layer provides a suite of general purpose policies. These policies represent those commonly provided by secure group communication systems. Clearly, there are many policies beyond what is available in this layer. Where required, an application may extend or replace these policies by accessing di-

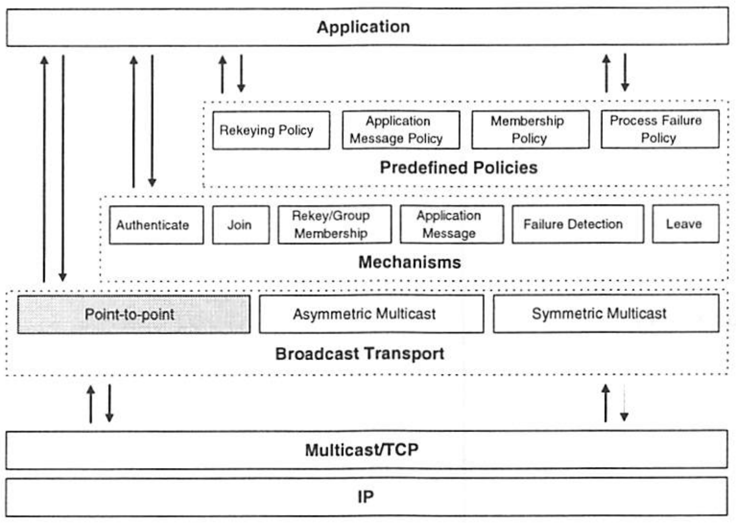


Figure 1: The architecture of Antigone consists of three layers; the broadcast transport layer, the mechanism layer, and the predefined policy layer. The broadcast transport component provides a single broadcast abstraction for environments with differing levels of support for multicast. The mechanism layer provides a set of primitives used to implement application policies. The predefined policies layer provides a suite of general purpose policies. Where required, applications may define other policies by accessing the mechanism and broadcast transport layer directly.

rectly the broadcast transport and mechanisms layers. We describe how the predefined polices are implemented through the Antigone mechanisms in Section 6.

5 Mechanisms

We chose a composite protocol approach [HS98] for the design of Antigone mechanisms. In composite protocols, features are partitioned into modules consisting of shared state, messages, events, and micro-protocols. This approach has several desirable properties that make it applicable to Antigone. First, the modular design allows for the integration of future enhancements with minimal effect on other modules. Second, the micro-protocol approach allows state sharing between modules. We avoid the costs typically associated with state sharing between protocols (message headers).

Antigone provides mechanisms for providing the following functions; authentication, member join, session key and group membership distribution, application messaging, failure detection, and member leave. The Antigone micro-protocol variants associated with each mechanism are defined in Fig. 2. Before describing the micro-protocols, we explain the principals in the protocols and the notation used.

Fig. 3 shows the principals in an Antigone logical group. A distinct member of the group whose identity is known in advance to all members, called the *session leader* (SL), is the arbiter of group operations such as group joins, leaves, etc. We chose an arbitrated group because of its low cost and its applicability to many applications. For example, in a secure pay-per-view video broadcast application, the cable company would provide a session leader that enforces the desired access control and key distribution policy.

A *Trusted Third Party* (TTP) provides the mechanism for the session leader to authenticate joining *group members*. Each potential member, A, of a group (including the session leader) has a shared secret K_A registered with the trusted third party (TTP). This secret key is generated and registered with the *TTP* before the party attempts to join any session. We assume an out of band method for registering these keys and for informing everyone about the identity of the session leader.

In our protocol descriptions, we use the term *SL* to refer to the identity of the session leader, A to refer to a current or potential member of the session, and *TTP* to refer to the trusted third party. $\{X\}_k$ refers to message *X* encrypted under the key *k*. The view identifier, *g*, is used to uniquely tag the changing views of group membership. The term SK_g refers to a session key in view g, and SK_{g+1} for the (next) view $g + 1$. The term *I*, possibly with a subscript, refers to a nonce value. Key distribution protocols based on Leighton-Micali define a term $\pi_{A,B}$, called a *pair key*, used to support secret communication

Authenticate

1. $A \to SL : A, I_0$ *(authentication request)*
2. $SL \to TTP : SL, A, I_1$ *(pair key request)*
3. $TTP \to SL : \{[\pi_{SL,A} = \{A\}_{K_{SL}} \oplus \{SL\}_{K_A}], I_1\}_{K_{SL}}$ *(pair key response)*
4. $SL \to A : SL, A, \{g, A, I_0, I_2, [policy\ block], Pu_G\}_{\sigma_{SL,A}}$ *(authentication response)*

Join

5. $A \to SL : A, \{A, I_2\}_{\sigma_{SL,A}}$ *(join request)*

Rekey/Group Membership

6a. $SL \to A : g, S_{SL}, (A, \{g, SK_g\}_{\sigma_{SL,A}})\{H(g, S_{SL}, (A, \{g, SK_g\}_{\sigma_{SL,A}}))\}_{SK_g}$ *(key distribution)*

6b. $SL \to A : g, S_{SL}, (A, \{g, SK_g\}_{\sigma_{SL,A}}), B, C, D, \ldots, \{H(g, S_{SL}, (A, \{g, SK_g\}_{\sigma_{SL,A}}), B, C, D, \ldots)\}_{SK_g}$ *(key/group membership distribution)*

6c. $SL \to group : g+1, S_{SL}, (A, \{g+1, SK_{g+1}\}_{\sigma_{SL,A}}), (B, \{g+1, SK_{g+1}\}_{\sigma_{SL,B}}), \ldots,$
$\{H(g, S_{SL}, (A, \{g+1, SK_{g+1}\}_{\sigma_{SL,A}}), \ldots)\}_{SK_{g+1}}$ *(session rekey)*

Application Messaging

7a. $A \to group : g, A, [message], \{H(g, A, message)\}_{SK_g}$ *(with integrity)*
7b. $A \to group : g, \{A, [message]\}_{SK_g}$ *(with confidentiality)*
7c. $A \to group : g, \{A, [message]\}_{SK_g}, \{H(g, \{A, [message]\}_{SK_g})\}_{SK_g}$ *(with integrity and confidentiality)*
7d. $A \to group : g, A, [message], H(g, A, message)_{C_A}$ *(with sender authenticity)*

Failure Detection

8. $A \to SL : g, A, S_A, H(g, S_A)_{\sigma_{SL,A}}$ *(member heartbeat)*
9. $SL \to group : g, SL, S_{SL}, H(g, S_{SL})_{Pr_G}$ *(session leader heartbeat)*
10. $A \to SL : g, A$ *(key retransmit message)*

Leave

11. $A \to SL : A, \{g, A, \{g, B\}_{SK_g}\}_{\sigma_{SL,A}}$ *(leave request)*

Figure 2: Antigone Micro-Protocol Description - micro-protocols for the various operating modes. A *composite protocol* is constructed from the selection of a subset of these modes. In some configurations, some micro-protocols may be omitted entirely.

between collaborating members A and B. Derived from the pair key, the session leader and a potential member A maintain a shared secret key $\sigma_{SL,A}$. A cryptographic hash for the text x is described by $H(x)$. We use the MD5 hash algorithm [Riv92] in our implementation.

The format of the identity, nonce, key, and view identifier values used in our current protocol implementation is as follows. Each identity is a unique 16 byte null terminated ASCII string of alphanumeric characters. A potential member is assigned this value when registering a long term key with the *TTP*. Nonce values are unique 64 bit values. To ensure that nonces are not reused, some source of monotonic values, such as the system clock, may be used. Key format is algorithm dependent. The DES standard uses an eight byte key (including eight parity bits). In the future, as other ciphers are integrated into Antigone, we will need to support other formats. The view identifier g consists of a two parts. The first part is an eight byte, null terminated name string that identifies the group, used only for displaying and debugging purposes. The second is an eight bit nonce value. Each time a new view is created ($g + 1$), a new nonce is generated and appended to the group name string to create the new view identifier.

A *policy block* is distributed by the session leader to each group member during the authentication process. Defined by the application, the policy block is an arbitrary byte string stating the group policy. We describe the use of this data by the predefined policies layer in Section 6.

The session leader creates an *asymmetric key pair* (Pu_G, Pr_G) during group initialization. The public key (Pu_G) is delivered to potential members during the authentication process. The public key is used to reduce the cost of sending secure heartbeats from the session leader.

Where sender authenticity is configured, we assume the existence of a certificate distribution service [HFPS98]. The certificate service would provide access to certificates for each group member (C_A). Note that no certificate distribution service is required in our protocol to generate or distribute the (Pu_G, Pr_G) asymmetric key pair.

We use DES [Nat77] for all encryption in the current implementation. Its inherent strength is evident from its 20-year history, yet its 56-bit key length has long been the subject of debate. Related algorithms such as triple-DES [ANS85] or DESX [KR96] offer the strength of DES with considerably longer keys. Our protocols are not tied to any specific property of DES, and may be replaced with other cryptographic algorithms as necessary.

We assume that all processes that have achieved membership, and thus have been authenticated, adhere to the system specification. We assume no member willingfully discloses its long term or session keys. All members trust the *TTP* not to disclose their long term key, and to generate pair keys according to the specification.

The following text describes each of the micro-protocol modules in Fig. 2. All cited message numbers refer to Fig. 2.

Authentication Mechanism - The authentication mechanism provides facilities for the authentication of potential group members. The purpose of this mechanism is twofold. First, the session leader authenticates the potential group member. Second, a shared secret between the two parties is negotiated. The shared secret, called the *shared secret key*, is later used to implement a secure channel between the two parties.

We use the Leighton-Micali key distribution algorithm [LM94] to authenticate the joining process and negotiate the shared secret. The main advantage of Leighton-Micali is low cost; it uses symmetric key encryption throughout, with none of the modular exponentiation operations associated with public key cryptosystems. Public key cryptography requires significantly more computation than symmetric algorithms. The de-facto standard for public-key cryptography, RSA, can be up to 100 times slower in software and 1000 times slower in hardware than DES, the predominant symmetric algorithm [Sch96].

A prospective member initiates the authentication process by sending a message to the session leader containing her identity and a nonce value (message 1). The session leader then obtains the pair key $\pi_{A,B}$ from the *TTP* (messages 2 and 3). Derived from two identities and their associated long term keys, the pair key is used to establish an ephemeral secure channel between the processes. To prevent replay attacks, the session leader verifies the encrypted nonce value I_1 included in the *TTP*'s response.

The session leader computes the shared secret key for communicating with *A* as follows. The session leader generates the value $\{A\}_{K_{SL}}$. This value is XOR-ed with the pair key $\pi_{SL,A}$ received from the *TTP*. The resulting value is a shared secret key ($\{SL\}_{K_A} = \sigma_{SL,A}$) that is used to create a secure channel between the session leader and the prospective member *A*.

Note that *A* need not communicate with the *TTP* to obtain the shared secret key $\{SL\}_{K_A} = \sigma_{SL,A}$; she can compute it directly. *A* decrypts the session key with this value and validates her nonce.

After obtaining the shared secret key, the session leader responds with an authentication response message (message 4). The response contains the identities of the session leader and the potential group member, and a block encrypted with the shared secret key $\sigma_{SL,A}$. The encrypted block contains the view (g) and group member (A) identifiers, the group member nonce (I_0), a session leader nonce (I_2), the policy block, and the group public key (Pu_G). Upon receiving this message, the receiver decrypts the contents and verifies the nonce I_0.

Join Mechanism - The join mechanism provides a member with facilities for gaining access to the group. The mechanism also provides measures to ensure the reliability of the join.

The potential group member (A) joins the group by transmitting message 5 to the session leader. Upon reception of message 5, the session leader validates the nonce value (I_2) passed to the joining member during the authentication process. If the nonce is not valid, the join request is ignored, and the group continues. If the nonce is valid, the new member is accepted into the group. The reaction of the session leader to the join request depends on the configured group model (see below).

Note that the join message is unforgeable and fresh. A session leader knows that a correct join message is fresh and was generated by the member because of the presence of the (I_2) nonce value encrypted under the shared secret key.

Mutual authentication is achieved through the verification of the secrets *A* and *SL* share with the *TTP*. The potential member must be in possession of the secret shared with the *TTP* to obtain the session leader nonce (I_2) used in joining the group in message 5. The session leader must be in possession of the secret shared with the *TTP* to determine the secret key shared with *A*. *A* is convinced that message 4 is fresh by validating the nonce

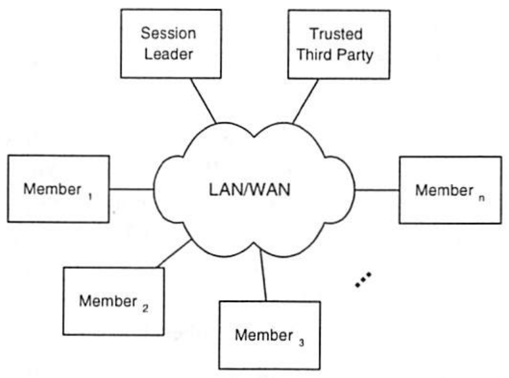


Figure 3: An Antigone group consists of an arbiter called the *session leader*, a service providing member authenticity called the *trusted third party*, and the group members. No assumptions are made about the network topology or connectivity.

value I_0 sent in the original request.

Rekey / Group Membership Mechanism - The Rekey/Group Membership mechanism provides for the distribution of group membership and session keys. We note the distinction between *session rekeying* and *session key distribution*. In session rekeying, all existing group members must receive a newly created session key. In session key distribution, the session leader transmits an existing session key.

The rekey and session key and distribution messages (6a, 6b, and 6c) all contain a group identifier (g), the latest session leader sequence number (S_{SL}), and a *Message Authentication Code* (MAC) calculated over the entire message, $H(\ldots)$. The group identifier and sequence number identify the current group context. The MAC ensures integrity of the message.

Session keys are distributed via *session key blocks* $(A, \{g, SK_g\}_{\sigma_{SL,A}})$. The intended member of each block is identified by the group member identifier (A). The remainder of the block is encrypted using the shared secret key ($\sigma_{SL,A}$). If the group identifier is decrypted by the receiver correctly (matches the identifier in the message header), the member is assured that the block was created by the session leader.

Message 6a contains a session key block for one member, and no group membership information. Message 6b contains a session key block for one member and enumerates the current group membership ($B, C, D, \ldots$). In message 6c, a session key block is generated for each member in the group. Group membership in message 6c is extracted from the session key blocks.

Rekeying is performed by the transmission of message 6c. Rekeying in Antigone is similar to key distribution after a LEAVE operation in the Iolus system [Mit97]. The session leader caches the shared secret keys, so creating this message is fast: encryption of 24 bytes (8 bytes of new session key plus 16 bytes of new group identifier) per member. Using the cached, shared secret key, the receivers of this message extract the session key out of the session key block and begin using it immediately. The size of this message grows linearly with group size, and is potentially large. In systems that provide session key independence, keying material needs to be sent to each member individually. Therefore, the size of the message is large by its nature, not as a side effect of our design. Schemes that distribute a key to each member individually will transmit the same amount of data over many more messages.

A potential problem occurs during the transition of group views. During the rekeying process, application data such as continuous media may continue to be broadcast using a previous session key. Because of delays in the delivery of the session key, a process may receive a message encrypted with a session key that it does not yet or will never possess. An application may address this issue by buffering, double encryption, dropping packets, or other approaches. We present a detailed discussion the positive and negative aspects of several of these approaches in [MHP98].

We are in the initial stages of implementing a key hierarchy based rekeying mechanism (see Section 3). It has been shown that the costs of rekeying may be vastly reduced using this approach. In the future, we will study the effect of reduced rekeying costs on policy.

Application Messaging Mechanism - The application messaging mechanism provides for the transmission of application level traffic. Each application level message is secured using cryptographic keys distributed by the rekey/group membership mechanism, or through the use of external public key certificates.

The format of application messages is dependent on the type of messaging policy. We achieve message integrity through *Message Authentication Codes* (MAC) [Sch96], and confidentiality by encrypting under the session key. Message 7a shows the format of a message with integrity only, message 7b shows confidentiality only, and 7c shows a message with both integrity and confidentiality.

A MAC is generated by encrypting an MD5 hash [Riv92] of the message data under the session key. A receiver confirms the MAC by decrypting and verifying the MD5 hash value. If the hash is correct, the receiver is assured that message has not been modified by some entity external to the group.

Sender authenticity (message 7d) is achieved by digital signature [DH76]. The signature is generated using the private key exponent associated with the sender's certificate. Receivers obtain the sender's certificate and verify the signature using the associated public key. Note that a byproduct of the use of digital signatures is message integrity. Our current implementation does not support sender authenticity.

Failure Detection Mechanism - An application's threat model may require the system to tolerate attacks in which an adversary prevents delivery of rekeying messages from the session leader to the entire group or to a subset of members. In such a case, some members will continue to use an old session key, a security risk if the old key is compromised. Also, for accurate membership information, it may be necessary for the session leader to be able to detect fail-stop failures of members.

In Antigone, we provide *secure heartbeat* messages as the mechanism to detect failed processes. The session leader detects failed processes through member heartbeats (message 8). When some number of member heartbeat messages are not received by the session leader, the member is assumed failed and expelled from the group.

Group members confirm that the session leader is still operating by receiving the session leader's secure heartbeat messages (message 9). When some number (the value of which is a policy issue for higher layers) of session leader heartbeat messages are not received, the member can assume that the session leader has failed.

Heartbeat messages serve a dual purpose. In addition to failure detection, members use the heartbeats to ensure that they have the most current group state. The session leader's sequence number ensures that the heartbeat is fresh. The presence of the group identifier allows a group member to be certain that they are using the most recent session key. The heartbeats are encrypted to ensure that an adversary cannot fake heartbeats. Without these protections, an adversary may be able to prevent delivery of new session keys and trick members into continuing to transmit under an old session key indefinitely.

A member who fails to receive a current session key or membership information can attempt to recover by sending a key retransmit message (message 10). The key retransmit message indicates to the session leader that the member wishes to get the most recent group state. The session leader will send the most key/group membership distribution (message 6a, 6b, or 6c) in response to the key retransmit message. In this case, the process will be able to recover by installing the most recent session key and/or group membership.

It is worth noting that congestion that causes rekey or heartbeat message loss may be exacerbated by the resulting requests for more recent group state. This problem, known as *sender implosion*, is likely to limit the efficiency of Antigone in large groups or on lossy networks. We plan to introduce a retransmit mechanism similar to SRM [FJL$^+$97] to address this limitation. Using this approach, we distribute the cost of retransmission among the group by allowing any member to respond to requests for lost messages. Retransmission requests are made a random time after the loss is detected (the random time is computed as a function of measured distance from the session leader). Members noting a retransmission request suppress local requests, and wait until a retransmission is received or a timeout occurs. Conversely, members receiving retransmission requests for data (heartbeat or key messages) they have received wait a random time (which is a function of the distance from the requester) before retransmitting. If it is noted that some other member has performed the retransmission, the request is ignored. Note that this approach in no way affects the security of Antigone, but only serves to reduce the cost of retransmission request processing.

The Antigone infrastructure's goal at this level only provides mechanisms for reliable detection of session leader's failure and not recovery from its failure. Mechanisms for detection of failures can, if desired, be used to implement recovery algorithms using primary backup or replicated approaches at higher levels.

Leave Mechanism - This mechanism provides an interface for group members to gracefully exit the group. A member sends message 11 to indicate that it is exiting the group.

The leave mechanism has a secondary purpose. Using the micro-protocol defined in Fig. 2, a group member may request the ejection of other members from the group. To request an ejection, the requester places the identity of the member in the $\{g, B\}_{SK_g}$ block (as B). The session leader receiving a message with this format will eject the member in accordance with some local policy.

6 Policy Implementation

In this section, we show how flexible application policies may be implemented through the Antigone mechanisms.

Membership Awareness Policy

If members should not explicitly be given the list of cur-

Policy	JOIN	LEAVE	FAILURE	EJECT
Time Sensitive	N	N	N	N
Leave Sensitive	N	Y	Y	Y
Join Sensitive	Y	N	N	Y
Membership Sensitive	Y	Y	Y	Y

Table 1: The predefined rekeying policies may be defined by their sensitivity to membership events. Note that join sensitive groups are MEMBER_EJECT sensitive to allow for member ejection.

rent members in a group, then rekeying can be done via point-to-point messages to each member using message 6a. Otherwise, using message 6c via a multicast to rekey a group is generally more efficient. In either case, as pointed out in Section 5, we do not guarantee confidentiality of group membership against adversaries who are able to monitor network traffic and analyze packet flows in the group.

When a member fails to receive a rekey message, it can request a re-broadcast by sending message 10. If membership awareness is required in the group, the session leader can use message 6b or 6c to update the member; otherwise, the session leader can use the smaller message 6a to rekey the member, which is more efficient to encrypt and send.

Rekeying Policy Implementations

To implement a time-sensitive group, the session leader simply creates a new group identifier periodically, followed by group rekeying as describe above.

To implement a join-sensitive group, after a member joins (message 5), the session leader rekeys the session using either message 6a to each member or via message 6c.

If the policy is not join-sensitive, when a member joins, no rekeying is necessary. The new member is sent message 6b or 6a, depending on whether it needs to be provided the group membership list or not, respectively.

To implement a leave-sensitive group, when any member leaves (message 11), is ejected (message 11), or fails (detected via secure heartbeats), the session leader rekeys the group. The session leader rekeys the session by sending message 6a to each member or by multicasting message 6c to the group, depending on the membership awareness policy.

To implement a membership-sensitive group, session rekeying is done after joins, leaves, failures and ejections.

Application developers can implement other rekeying policies, given the mechanisms in the previous section. For example, the session leader can rekey only when certain members join or leave member in the group, or do time-sensitive rekeying in combination with membership-sensitive rekeying.

Application Message Policy Implementations

Several choices with different guarantees are available for sending messages (choices 7a, 7b, 7c, and 7d). It is up to the application to make judicious use of these available choices, depending on the requirements. We will discuss the performance implications of application message policies in the Performance section.

Predefined Policies

To simplify application development, Antigone provides a simple specification interface to select a group security policy. This specification interface allows selection from several common policies. Applications that require custom policies can, of course, implement their own policies directly though the Antigone mechanisms.

The predefined-policy layer in Fig. 1 uses the policy block in message 4 (see Fig 2) to send six fields (GG, SG, RK, HB, MM, and FP) to select a policy from a range of common policies.

The GG parameter is used to select from four implemented policies. The policies (TIME_SENS, LEAVE_SENS, JOIN_SENS, and MEMBER_SENS) correspond to the definitions presented in Section 2. Table 1 describes these policies in terms of their sensitivity to membership events.

The RK parameter is used to specify the rekey timer value. Each member resets the timer to the specified value when a new key is received. If the timer expires, the member considers the current session key to have expired and requests that the session leader send a new session key. The session leader should normally rekey a session prior to the expiration of this timer. Note that the combination of GG and RK parameters allow specification of rekeying policies that are both membership-sensitive as well as time-sensitive.

The SG parameter is used to specify the security guarantee on application messages. The guarantees that can be specified are: *confidentiality*, *integrity*, and *sender authenticity*. A side effect of the selection of any of these guarantees is that all application messages will have the *group authenticity* property. The group authenticity property states that messages can be verified to determine if they emanated from a member of the group. We outline the performance issues relating to these policies in Section 8.

The MM parameter is used to indicate if membership awareness is required by group members. If membership information is to be supported, it is distributed during every rekeying operation via message 6c or, in case of retransmit requests, via message 6b. Otherwise, message 6a is used to rekey members.

The failure policy (FP parameter) indicates whether failures of the session leader and the members are to be detected. If the process failure policy states failures are to be detected, heartbeat messages (9,10) are transmitted periodically. The failure detection mechanism will detect failed members as defined in the previous section.

The HB parameter is used to specify the periodicity of heartbeat messages, if any.

In general, if an adversary can prevent messages from being delivered, periodic heartbeats from the session leader are important. Since the session leader's heartbeats carry the group identifier and a sequence number, they help ensure that a group can be forced to use an old session key only for a period implied by the heartbeat interval.

Heartbeats are useful even when a combination of membership-sensitive and time-sensitive rekeying is used. For large groups, rekeying (message 6c) can be more expensive for the session leader compared to sending secure heartbeats (message 9). In such cases, the session leader can set the heartbeat interval to be lower than the rekey interval. The lower heartbeat interval ensures that members do not use an old key beyond that interval. In the absence of heartbeats, the rekey interval will define the bound for which an old key may be used.

7 Broadcast Transport Layer

Antigone provides three broadcast transport modes; *symmetric multicast*, *point-to-point*, and *asymmetric multicast*. In providing a single transport abstraction, Antigone supports network environments with varying levels of support for multicast.

The symmetric multicast mode uses multicast to deliver all messages. Applications using this mode assume complete, bi-directional communication is available via multicast. In effect, there is no logical difference between this mode and direct multicast.

The point-to-point mode provides a broadcast service based solely on point-to-point communication. All message traffic intended for the group is unicast to the session leader, and relayed to each group member via UDP/IP. However, as each message must be transmitted to group members individually, bandwidth costs increase linearly with group size. In some applications, these costs may be prohibitive. For example, a group of even modest sizes would have difficultly in maintaining a video transmission with reasonable frame rates.

In [AAC+99], we describe our experiences with the deployment of a video application based on an early version of Antigone. The deployed system was to securely transmit video and audio of the September 1998 Internet 2 Member Meeting. The receivers (group members) were distributed at several institutions across the United States. While some of the receivers were able to gain access to the video stream, others were not. It was determined that the network could deliver multicast packets towards the receivers (group members), but multicast traffic in the reverse direction was not consistently available (towards the session leader). The problem was attributed to limitation in reverse routing of multicast packets. We present significant technical detail of this issue in [AAC+99].

These problems coupled with the costs associated with a completely point-to-point solution lead us to introduce *asymmetric multicast*. This mode allows for messages emanating from the session leader to be multicast, and member messages to be delivered point-to-point. Members wishing to transmit a message to the group send the message to the session leader as a unicast. The session leader acts as a relay for all group communication. Any message intended for the group received by the session leader is re-transmitted via multicast. Thus, we reduce the costs associated with group delivery of a point-to-point solution to a unicast followed by a multicast.

Significant technical and administrative issues must be resolved before multicast services can be widely deployed in the Internet. In the interim, techniques such as asymmetric multicasting will be useful in the construction of group based systems.

The key advantage of a single abstraction is in its ability to handle updates in the networking environment. As symmetric multicast becomes universally available, neither Antigone nor applications built on it will require redesign to make use of the newly available service.

8 Performance

In this section we present a preliminary performance study of the Antigone framework. We illustrate the CPU costs of group key operations at the session leader and identify the throughput and latency characteristics of application messaging.

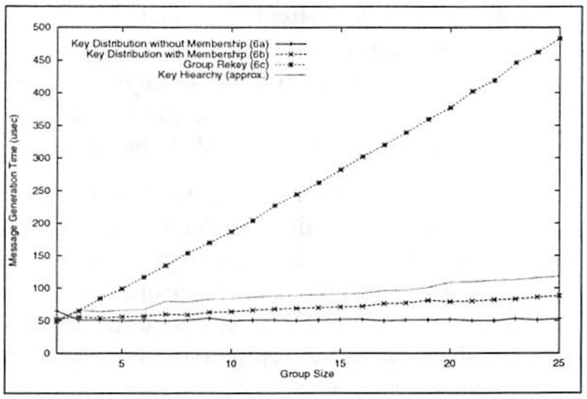


Figure 4: Group Membership/Session Key message generation costs - Measured and estimated costs associated with the message generation of groups of varying sizes and semantics.

The experiments described in this section were performed on unloaded Intel 200MHz Pentium Pro workstations running FreeBSD kernel version 3.0. All tests were executed on an unloaded 100 MBit Ethernet LAN. Throughput and latency experiments included a single sender (which was the session leader) and nine receivers.

Fig. 4 shows the cost of group membership/session key message generation under varying group models and sizes. For comparison, we estimate the message generation costs associated with a key hierarchy approach.

Generation of the session key distribution message without group membership information (message 6a) should be constant under all group sizes. Creation of this message includes the generation of one session key distribution block and the generation of a *Message Authentication Code* (MAC). The figure indicates a constant cost of message generation.

The costs associated with key distribution with group membership message (6b) generation increases slightly with group size. This trend can be attributed to the increasing amount of data to be hashed. As the group membership grows, the costs associated with MAC generation increase.

The costs associated with the rekey message (6c) increases linearly with group size. A session key block is generated for each member, which requires a distinct cryptographic operation. Similar to the message 6b, the cost of MAC generation increases with group size. Because of the speed of the underlying cryptographic algorithms, increases in the cost of message generation due to MAC construction is significantly less than increases due to session key block construction.

Policy	Throughput	Latency
Integrity	2577 KB/sec	5710 usec
Confidentiality	1697 KB/sec	8698 usec
Integrity and Confidentiality	1577 KB/sec	9037 usec

Table 2: Application Messaging Performance - Maximum throughput by an application sending 1 KB messages and latency of end to end delivery for a 10 KB message.

The Key Hierarchy data in Figure 4 estimates the cost of rekey message with membership (similar to 6c) generation in a key hierarchy based group. In a key hierarchy, rekeying is performed by the distribution a number of keys which is roughly proportional to the log of the group size. Therefore, we see little increase in message cost as the group becomes larger. This realization further indicates that a key hierarchy based rekeying mechanism will be a valuable extension to Antigone.

In Table 2, we show the throughput and latency characteristics of application messaging under varying security policies. Although not surprising, our experiments show that as the number of guarantees increase, so do the costs. Thus, policy has a direct affect on performance: stronger policies have less throughput and greater latency. There is a natural ordering to these guarantees. *INTEGRITY* (message 7a) is least expensive, *CONFIDENTIALITY* (7b) is more expensive, and the *INTEGRITY* and *CONFIDENTIALITY* (7c) is the most expensive.[2]

The ordering of application message generation costs mirrors the speed of the underlying cryptographic operations. We use DES [Nat77] to achieve confidentiality, and MD5 [Riv92] to achieve integrity. Our implementation uses the SSLeay v0.9.0b [HY98] `crypto` library. On the test machine, we found that DES encryption ($\approx$ 3.7 Mbyte/second) is about $1/6$ the speed of MD5 hashing ($\approx$ 21 Mbyte/second).

9 Conclusions and Future Work

In this paper, we presented the Antigone framework. Antigone provides a flexible interface for the definition and implementation of a wide range of secure group policies. Applications implement policy by composing

[2]Currently, we have not implemented the sender authenticity guarantee (message 7d), and thus do not show its cost. As it requires private-key encryption, we expect sender authenticity to be the most costly guarantee to provide.

and configuring secure group primitives called *mechanisms*. Thus, Antigone does not dictate the available policies, but provides facilities for building them.

The mechanisms provided by Antigone represent the set of features required to implement a secure group. The mechanisms include facilities for authentication, member joins, session key and group membership distribution, application messaging, failure detection, and member leaves. Policy-implementing software use these mechanisms to construct a feature set specific to the application context and the assumed threat model.

The target applications of Antigone require a low-cost solution. A result of this requirement is that the mechanisms in Antigone provide simple, but substantive, features for implementing various security policies.

To validate the primitives as well as to simplify development of applications, we have constructed a suite of general-purpose security policies. These predefined policies represent those that have been found useful or have been suggested as being useful by various group communication systems.

Though not typically associated with secure group communication services, Antigone provides an abstract interface for multicasting. A reality of the existing Internet fabric is its inconsistent support for multicast services. In deploying multicast-based solutions, we have found that though multicast connectivity in one direction is often possible, achieving bi-directional multicast is more difficult. As a result, we introduce a transport mode called *asymmetric multicasting*. In asymmetric multicasting, messages emanating from a single source use multicast, and all others use unicast. Antigone's implementation provides interfaces for symmetric multicast (bi-directional), asymmetric multicast, and point-to-point (unicast) group communication.

Our initial performance study indicates that as policy requirements increase, so do the performance costs. This is not a surprising result, but indicates the need for security infrastructures to support a range of security policies. In the near future, we will extend this study to include a more thorough analysis of latency, throughput, and scalability characteristics of Antigone groups within several networking environments.

An early version of Antigone has been integrated into the the `vic` Videoconferencing application [Net96]. A number of policy issues arose during the implementation and deployment of the resulting *Secure Distributed Virtual Conferencing* (SDVC) system [AAC+99]. The current design of Antigone represents the many refinements prompted by the analysis of SDVC.

Several challenges remain. Applications may have requirements for greater fault tolerance. The need for services that provide greater scalability is evident. In the future, we hope to investigate ways to meet such requirements, while retaining simple mechanisms that support flexible security requirements.

10 Availability

Source and documentation for the Antigone system are available from

`http : //antigone.citi.umich.edu/.`

11 Acknowledgements

We would like to thank William A. (Andy) Adamson, Charles J. Antonelli, and Kevin Coffman for their assistance in the development of the Antigone system, Weston A. (Andy) Adamson for his help in constructing and maintaining the Antigone web-site, and the anonymous reviewers for their many helpful comments.

References

[AAC$^+$99] A. Adamson, C.J. Antonelli, K.W. Coffman, P. D. McDaniel, and J. Rees. "Secure Distributed Virtual Conferencing". In *Communications and Multimedia Security (CMS '99)*, September 1999.

[ANS85] ANSI. "American National Standard for Financial Institution Key Management". *American Bankers Association*, 1985. ANSI X.917.

[Bal96] A. Ballardie. "Scalable Multicast Key Distribution". *Internet Engineering Task Force*, May 1996. RFC 1949.

[BFC93] T. Ballardie, P. Francis, and J. Crowcroft. "Core Based Trees (CBT)". In *Proceedings of ACM SIGCOMM '93*, pages 85–95. IEEE, September 1993.

[Bir93] K. Birman. "The Process Group Approach to Reliable Distributed Computing". *Communications of the ACM*, 36(12):37–53, December 1993.

[Dee89] S. Deering. "Host Extensions for IP Multicasting". *Internet Engineering Task Force*, August 1989. RFC 1112.

[DH76] W. Diffie and M.E. Hellman. "New Directions in Cryptography". *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[FJL$^+$97] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing". *IEEE/ACM Transactions on Networking*, pages 784–803, December 1997.

[Gon96] L. Gong. "Enclaves: Enabling Secure Collaboration over the Internet". In *Proceedings of 6th USENIX UNIX Security Symposium*, pages 149–159. USENIX Association, July 1996.

[HFPS98] R. Housley, W. Ford, W. Polk, and D. Solo. "Internet X.509 Public Key Infrastructure, Certificate and CRL Profile". *Internet Engineering Task Force*, June 1998. (draft-ietf-pkix-ipki-part1-08.txt).

[HM97] H. Harney and C. Muckenhirn. "Group Key Management Protocol (GKMP) Architecture". *Internet Engineering Task Force*, July 1997. RFC 2094.

[HP94] N.C. Hutchinson and L.L. Peterson. "The x-Kernel: An Architecture for Implementing Network Protocols". *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1994.

[HS98] M. Hiltunen and R. Schlichting. "A Configurable Membership Service". *IEEE Transactions on Computers*, 47(5):573–586, May 1998.

[HY98] T. Hudson and E. Young. SSLeay and SSLapps FAQ, September 1998. http://psych.psy.uq.oz.au/ ftp/Crypto/.

[KA98] S. Kent and R. Atkinson. "Security Architecture for the Internet Protocol". *Internet Engineering Task Force*, November 1998. RFC 2401.

[KR96] J. Kilian and P. Rogaway. "How to Protect DES Against Exhaustive Key Search". In *Proceedings of Crypto '96*, pages 252–267, August 1996.

[LM94] T. Leighton and S. Micali. "Secret-key Agreement without Public-Key Cryptography". In *Proceedings of Crypto 93*, pages 456–479, August 1994.

[MHP98] P. McDaniel, P. Honeyman, and A. Prakash. "Lightweight Secure Group Communication". Technical Report 98-2, CITI, University of Michigan, April 1998.

[Mit97] S. Mittra. "Iolus: A Framework for Scalable Secure Multicasting". In *Proceedings of ACM SIGCOMM '97*, pages 277–278. ACM, September 1997.

[Mul93] Sape Mullender. *Distributed Systems*. Addison-Wesley, First edition, 1993.

[Nat77] National Bureau of Standards. "Data Encryption Standard". *Federal Information Processing Standards Publication*, 1977.

[Net96] Network Research Group, Lawrence Berkeley Laboratory. `vic` - Video Conferencing Tool, July 1996. http://www-nrg.ee.lbl.goc/vic/.

[RBM96] R. Van Renesse, K. Birman, and S. Maffeis. "Horus: A Flexible Group Communication System". *Communications of the ACM*, 39(4):76–83, April 1996.

[Rei94] M. Reiter. "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart". In *Proceedings of 2nd ACM Conference on Computer and Communications Security*, pages 68–80. ACM, November 1994.

[Riv92] R. Rivest. "The MD5 Message Digest Algorithm". *Internet Engineering Task Force*, April 1992. RFC 1321.

[Sch96] Bruce Schneier. *"Applied Cryptography"*. John Wiley & Sons, Inc., second edition, 1996.

[WGL98] C. K. Wong, M. Gouda, and S. S. Lam. "Secure Group Communication Using Key Graphs". In *Proceedings of ACM SIGCOMM '98*. ACM, September 1998.

[WHA98] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. Key Management for Multicast: Issues and Architectures *draft-wallner-key-arch-01.txt* (Draft). *Internet Engineering Task Force*, September 1998.

A Secure Station for Network Monitoring and Control

Vassilis Prevelakis
vp@unipi.gr
Network Management Centre
University of Piraeus, Greece

Abstract

The task of managing large networks is seldom accomplished using a single tool. Instead, network administrators usually compile collections of tools that can be used, in isolation or combined with others, to identify and correct problems. Equally important, however, is the platform used for the execution of those tools. In this paper we will be describing the Network Monitoring Station which has provided us with a safe base from which to troubleshoot network problems. This platform combines strong security with reduced configuration, administration and maintenance overheads. Moreover, it employs free, off-the-shelf software and runs on low-cost PCs.

1. Introduction

During the 80's, networks were smaller, but, most importantly, problems were accidental rather than intentional. The tools that were developed then, assumed that the entire user population had a stake in the correct operation of the network and associated resources. Nowadays, we find ourselves having to consider hostile action as a possibility when trying to resolve problems. This is extremely distracting, since, even now, most problems are still accidental. Nevertheless, being able to eliminate malice as the cause of an incident is a big step towards its successful resolution.

Another aspect of our heritage is that most network elements (switches, routers, etc.) have not been designed with strong security in mind. Even in production networks one can still gain useful information using the SNMPv1 public community. [Stal95] More worrying is that many such devices assume that administration can be carried out via a password protected telnet connection, despite the fact that this method has been shown to be insecure. [Garf96]

This project has been carried out within the EPEAEK series of programmes which are jointly funded by the Greek Ministry of Education and the European Community.

In the academic community, where a strong security policy is difficult, if not impossible to enforce, these problems are more evident. However, even in private companies or organisations, the use of firewalls and other security mechanisms is mostly directed towards outside threats, while leaving the interior network elements vulnerable to internal attacks. [Chap95]

Another handicap, shared by organisations outside the U.S, is that it difficult to get strong security out of the box on products that originate from the U.S. due to the well known export restrictions of cryptographic mechanisms.

To make matters worse, even in cases where vendors offer security features and strong authentication for their products, these are usually applicable only to the product range of the particular vendor. Thus, the configuration and management of the security features is a constant headache for the management team. [Shah97]

2. Requirements

At the University of Piraeus we realised that there was no point in trying to impose a global security policy because its implementation via firewalls and router access lists would not be acceptable to the local community. An earlier attempt to provide restricted services to dial-up users had to be abandoned because of user dissent.

In the end, we decided to retreat to higher ground by creating smaller networks within the larger campus area network that offer increased security. For example, the network management workgroup LAN is screened behind a firewall while key departmental servers, as well as the University Internet server are also kept in similar secure networks. In order to monitor activity outside these networks we decided to investigate the feasibility of producing a configuration for a personal computer (PC) that would enable us to create a vendor independent secure network monitoring station. These stations would be cheap and versatile so that they could be placed in various parts of our network and interfaced to different networking gear (see figure 1). The requirements for a machine of this class would be as follows:

- Low cost, preferably constructed from parts taken from decommissioned PCs.
- Minimal administrative overhead. This implied easy configuration and no administrator intervention after installation. Moreover, the bulk of the work for the construction of the software distribution for the network monitoring station should be devoted to integration of existing tools and packages, rather than the development of new code that would have to be maintained.
- Offer secure (encrypted) network connections with other similar stations and with the workstations of the network management staff.
- Be resistant to tampering. In the case where there are indications that the station has been hacked, its original configuration must be easily restored.
- Offer a standard platform for the execution of common network management and monitoring tools. It must also support the SNMP protocol.
- It must offer ways of establishing connections with network elements of various vendors for the purposes of administration and configuration.
- Finally, for troubleshooting purposes, it must be able to be deployed with minimal overheads in any part of the network.

In short, our intention was to construct something that could be used like meteorological balloons or sonar buoys: off-the-self and easily redeployable after use.

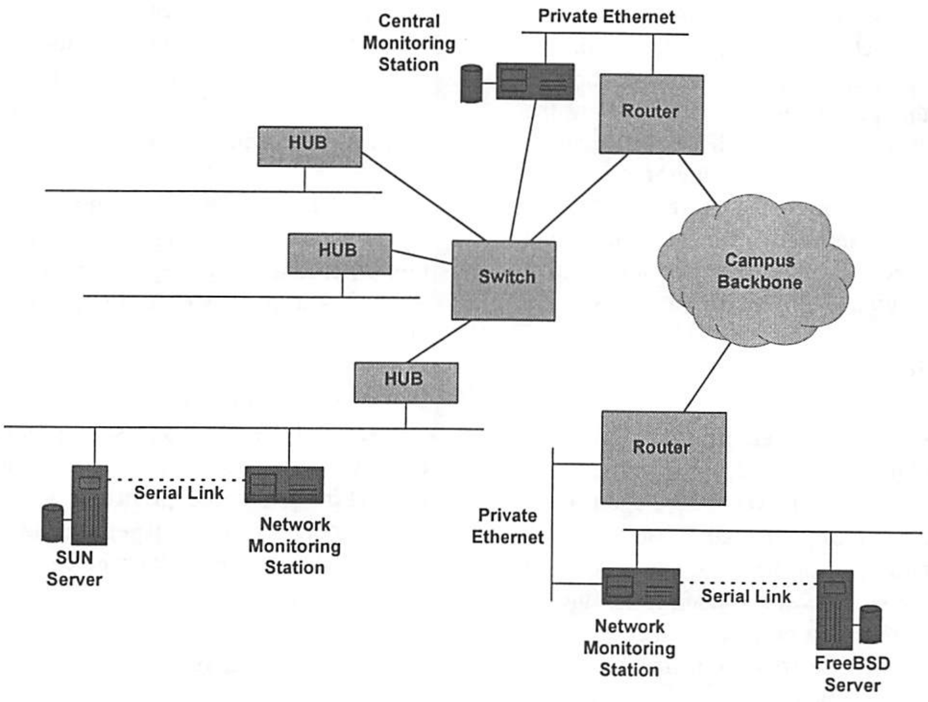


Figure 1: Network Monitoring Stations located in various parts of the network.

3. Network Monitoring Station

From the very beginning, the design team wanted a platform that could accommodate a large number of tools for network monitoring and management. The requirement that the station should operate in wiring closets without a monitor, keyboard or mouse effectively disqualified all Windows platforms. From the available UNIX or UNIX-like systems we eventually chose OpenBSD 2.3 for the following reasons:

- Like other free UNIX-clones, a large number of programs like tcpdump, snmpd, ssh, etc. are either supported in the base release or can be easily ported.

- The designers of OpenBSD have paid a lot of attention to the security profile of the system, creating a robust environment that is resistant to security related attacks (*http://www.openbsd.org/goals.html*)

- Most importantly, the system supports IPsec out of the box.

The next big decision that we took, was to dispense with a hard disk. The reason behind this decision was twofold, reliability and support. Older equipment, like the ones we use, tend to have problems with their hard drives, especially in the kind of hot environments that we let them operate. Greece is pretty hot in the summer and these PCs are left running in offices without air-conditioning for extended periods of time. Hard disks contribute a fair amount of heat and are also more prone to failure in these conditions.

The second and more important reason was related to the way that these machines were intended to be used. For our purposes, hard disks are already huge and are getting bigger all the time. This free space can cause all kinds of trouble; for example, it can be filled with data that should not be stored in the monitoring station in the first place. This means that stations can no longer be redeployed easily because this information must be backed up, or processed. Secondly, if a station is compromised, the intruders will be able to use this space as a bridgehead, transferring and installing tools that will enable them to attack other network assets.

On the other hand, diskless machines bring with them a whole collection of problems and administrative headaches. They are also basically incompatible with our intention of using standalone machines with encrypted tunnels for all communications between the monitoring stations.

Instead, we adopted the techniques used by the PICOBSD project which is a collection of FreeBSD configurations that can be accommodated within a single boot floppy (*http://www.freebsd.org/~picobsd*). The PICOBSD project provides configurations for a dial-up router, dial-in router (ISP access server), general purpose router and firewall. The PICOBSD technique links the code of all the executables that we wish to be available at runtime in a single executable using the *cruchgen* utility. [Silv98] The single executable alters its behaviour depending on the name under which it is run (argv[0]). By linking this executable to the names of the individual utilities we can create a fully functional /stand directory. The root of the runtime file system together with the executable and associated links are placed in a ramdisk that is stored within the kernel binary. The kernel is then compressed (using *gzip*) and placed on a bootable floppy. This floppy also contains the /etc directory of the running system in uncompressed form to allow easy configuration of the runtime parameters. At boot time, the kernel is copied from the floppy disk to main memory, uncompressed and executed. The file system root is then located in the ramdisk. The floppy disk is mounted and the /etc directory copied to the ramdisk. At this point the floppy is no longer needed and may be removed. The system is running entirely off the ramdisk and goes multi-user running the /etc/rc* scripts. Once the boot process is complete, user logins from the console or the network are allowed. The floppy is usually write-protected so changes in the system configuration do not survive reboots. However, there exists a utility that can copy the contents of the ramdisk /etc directory to the floppy, thus making the running configuration, permanent.

The aggregation of the system executables in a single file and the compression of the entire kernel allows a surprising number of facilities to be made available despite the small size of the boot medium. Some of these services are described below:

- **Accessing the system**
  The system establishes IPsec tunnels to a number of other secure hosts while shutting non-essential conventional IP ports through a packet filter (*ipf*). The initial configuration assumes statically assigned IP addresses and /etc/hosts table. Once the tunnels are running, a connection to the DNS server of the secure network may be established.

  Administrators on workstations that do not support IPsec (for example running Microsoft Windows 9x or NT) must rely on ssh for logging-on to the secure network hosts. This implies that the monitoring stations also run *sshd*. (*http://www.ssh.fi*). Over the IPsec links we run normal *telnet*.

  The Windows workstations run Teraterm which is a free telnet program that takes an ssh plug-in (*http://www.zip.com.au/~roca/ttssh.html*). Additionally, there is a commercial version of ssh for these platforms (*http://www.ssh.com*).

- **Logging**
  Since the systems do not have any permanent storage, we have to send the system logs to the central monitoring station. This is a conventional (with disk) OpenBSD system that has IPsec links to all the other stations. The transfer of logging information to the central station is performed using *syslogd* over the IPsec links.

- **SNMP agent**
  Another way of controlling the system is via the SNMP protocol. The UCD *snmpd* program (*ftp://ftp.ece.ucdavis.edu:/pub/snmp/ucd-snmp.tar.gz*) through a custom MIB, supports remote execution of commands as well as a number of status monitoring facilities.

- **Remote console agent**
  This agent allows serial ports to be accessed from other stations connected to the secure network.

Some configurations contain so many tools that they simply cannot fit within the boundaries of a single floppy. In these cases we use a CDROM that contains the standard OpenBSD /usr partition and we either copy the utilities we need to the ramdisk or we leave the CDROM mounted on /usr.

4. Examples of use

In this section we give two examples of the use of the system we have just described. The first example describes the way this system is used within the University network to manage routers and other network elements, while the second example describes the use of this system to handle the communication of network management teams of GUnet, a network linking Greek Universities.

4.1 Campus Network

The network monitoring stations have been deployed within the University of Piraeus network in three roles:

Controller: the monitoring station is connected directly to a router or other network device (e.g. switch, access server, etc.) so that configuration and administration of the device is carried out through the secure network. These connections can be serial links connecting one of the station built-in serial ports to the router console. Logins to the network device through its network ports are disabled so that administration can be carried out only via the console port. Hosts with serial consoles (e.g. SUNs) can also be controlled in this way.

An alternative means of connection is via back-to-back Ethernet links (see figure 2). In this case we dedicate an Ethernet port from the router and an

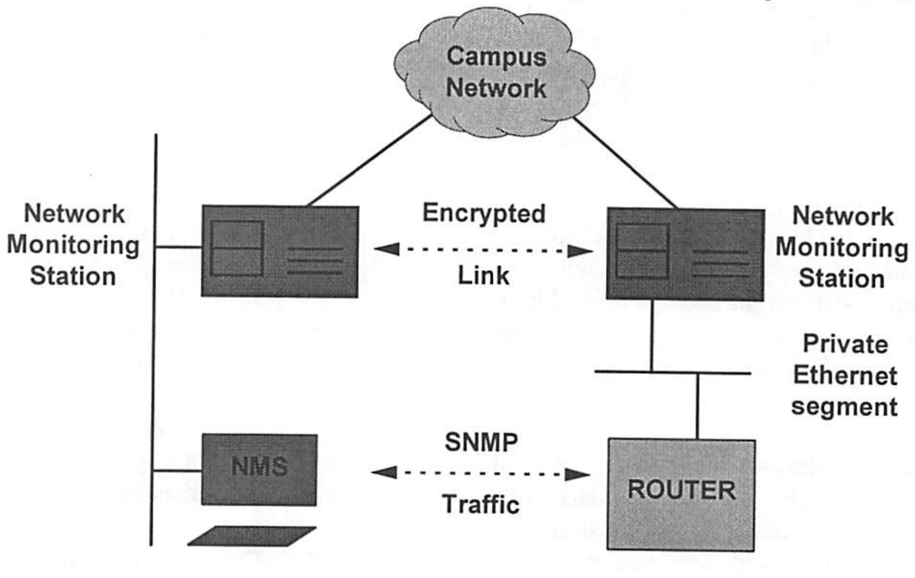


Figure 2: Patching external devices into the secure network

Ethernet port from the network monitoring station and link them via a crossed UTP cable. Via appropriate configuration of the router (e.g. ACLs) we can make sure that logins are allowed only from that Ethernet port. Similarly SNMP traffic may also be directed to that port. The Network Monitoring Station then relays the SNMP traffic through the IPsec links to the NMS workstation.

Traffic Monitor: We can monitor traffic on various LAN segments using *tcpdump* and send the output via *syslogd* to a central logging host. The syslog traffic goes over the IPsec links so that it cannot be intercepted (see figure 3)

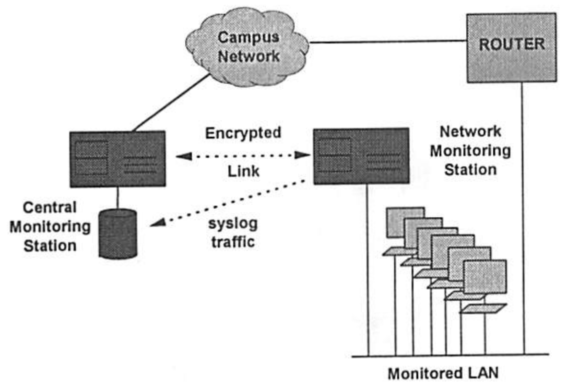


Figure 3: Monitoring remote parts of the LAN

Router: by adding high speed serial cards to the Network Monitoring Station we have created an emergency router for 2Mbps connections to buildings located outside the main campus and linked to the main building via leased lines (see figure 4). The OpenBSD kernel can support IP routing and through the addition of routing software (e.g. *gated*), it can exchange routing information (OSPF) with dedicated routers.

4.2 Monitoring stations in a WAN

Our second example concerns the Greek University Network (GUnet). In Greece, all public institutions of higher education have received funding in order to connect to a single high speed network (GUnet). Funding has also been provided for a local router, a server (in most cases a SUN workstation) and a network administrator stationed in each institution. These network administrators need to be in contact with the central network administration team in Athens.

Traditionally, these communications are carried out using PGP or some form of encrypted email scheme. However, for network monitoring purposes the administrators in the central site also need to be able to monitor the state of the routers located in the remote institutions. In cases of network problems, after consultation with the local network staff, the central site administrators may also need to run tests or reconfigure the GUnet routers and servers in the remote institutions. These operations would normally be run over insecure links, thus, creating opportunities for various exploits.

Due to the above considerations a small scale pilot

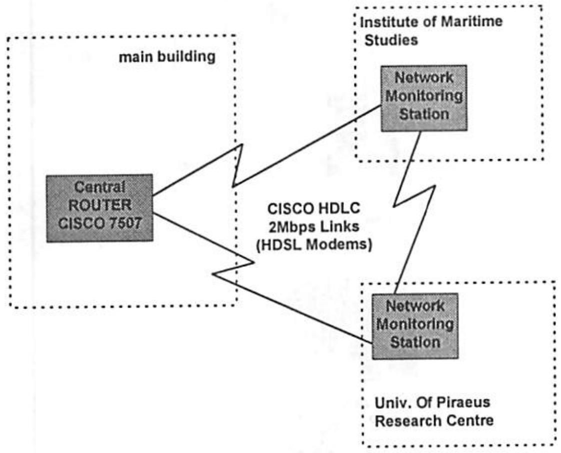

Institute of Maritime Studies
main building
Network Monitoring Station
Central ROUTER CISCO 7507
CISCO HDLC 2Mbps Links (HDSL Modems)
Network Monitoring Station
Univ. Of Piraeus Research Centre

Figure 4: Network Monitoring Stations may act as routers

has been commissioned by GUnet to investigate the possibility of employing network monitoring, stations similar to the ones mentioned in the previous section, in all the GUnet sites. The pilot project involves the installation of a central monitoring station at the University of Piraeus and remote stations in four other institutions.

The configuration of the remote stations is displayed in figure 5. The central site is hosting the logging facility and also provides email and bulletin board services (for CERT and other security advisories) only to hosts within the secure network.

The serial links to the routers and hosts allow remote reconfiguration of the devices through a secure connection via the IPsec links. Also using the backup WAN links that most institutions have with different ISPs (not shown in the diagram), the secure network can be used to debug even the GUnet routers via their serial consoles.

The pilot project is expected to be complete by the end of March 1999, at which point a study will be carried out to determine the feasibility of creating a production system covering all institutions participating in the GUnet project. If all goes well, the full system will go on-line by the end of this year.

5. Evaluation

The system within the University of Piraeus has been in operation for more than six months and is used to control most of the network elements that under the administration of the network management centre. The GUnet pilot has been in operation for about two months.

During this time we have acquired significant experience with respect to the behaviour of the systems and the needs of the administrators who use them. In this section we will discuss some of these issues.

5.1 Securing the perimeter

Setting up a collection of IPsec tunnels does not automatically create a VPN. Each IPsec endpoint should also act as a firewall. Thus packet filters must be configured using the `ipf` utility and in some cases application-level gateways must also be installed.

This is by necessity a very precise task, but, at least, the resulting configuration files can be used on all the stations. You cannot, for example, simply turn off IP packets and allow only AH or ESP packets through, because this will shutdown the loopback interface as well. Also some services must go through (e.g. ssh, snmp, etc) and you may want to be able to ping the secure hosts from the outside, so they should respond to ICMP packets. Finally, what use would a monitoring station have, if it could not see the monitored hosts and these machines are by definition outside the secure network perimeter.

Moreover, IPsec configuration is also complex, since the connections are unidirectional, thus requiring, two secure associations for each pair of hosts.

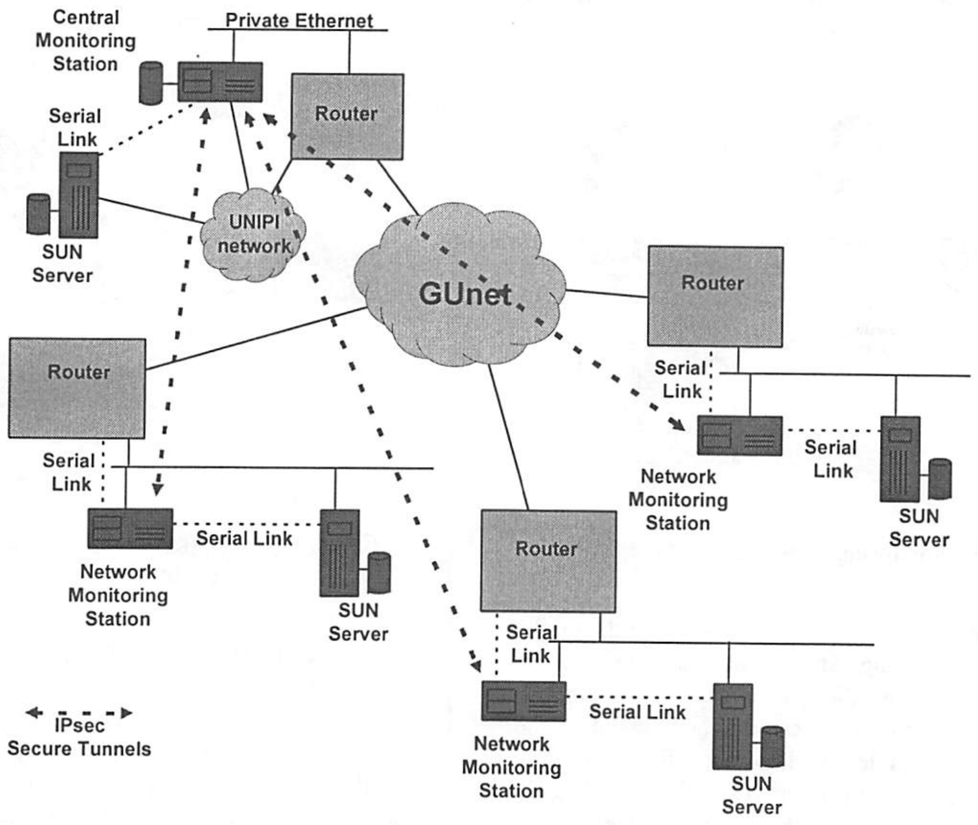


Figure 5: GUnet Network Monitoring Stations.

There are also a few pitfalls in configuring these tunnels. When we created the first few nodes we noticed that pinging one host resulted in the ICMP reply being sent in the clear (i.e. outside the IPsec link). This turned out to be a peculiarity of the IP stack whereby the system defers inserting the source address in a packet until the outgoing interface is determined. This way the packet gets the IP address of the correct network interface. The result of this is that the IPsec routing code needs an additional rule to catch such packets.

Using `tcpdump` to trace connections between two secure stations will often reveal holes either in the packet filter or the IPsec tunnels.

5.2 Automation

In the case of the GUnet pilot, we wanted to create a fully connected secure network, so that even if multiple nodes were disabled or locked out due to communication problems, the surviving nodes would still be able to communicate. The resulting mesh was likely to create a real mess because the number of tunnels exploded as the number of stations increased (2*n! to be exact).

Since it was clear that we couldn't possibly do this work manually, we wrote a set of shell scripts that created all the /etc configuration files from a single table containing information about each of the stations (e.g. name, IP address, netmask, etc.).

Each host was given a three digit number that was used for the generation of the secure associations. If hosts xxx and yyy wanted to set up a tunnel, then two SPI's would be assigned (xxxyyy and yyyxxx). Thus, the tunnels themselves could be easily identified and the scripts could generate the tunnel configurations automatically.

Similar scripts are used within the University of Piraeus network to create configurations for the diskette-based stations. In this case, all these stations are only connected to the main administration station which is also receiving their syslog records. Each subnet is given a colour code, so by typing `make blue`, we can create a standard floppy for the blue subnet.

5.3 Nostalgia

After entering the secure network the user is presented with an environment free from restrictions and checks. For example, the infamous r* utilities (`rsh, rwho, ruptime`, etc) all work without the need for specifying passwords. This is because we have deliberately made the assumption that the secure network should be considered as a logically contiguous secure region.

Obviously, this means that if an intruder gains access to a single host, then the entire network is compromised. This may be unacceptable in other environments, but in the case of our secure network, all the nodes are identical, so all nodes share the same vulnerabilities and hence a successful attack against one node will almost certainly succeed against any other node.

Nevertheless, the absence of internal barriers makes the network invisible to the user and thus improves the effectiveness of the administrators. For example, when was the last time you typed

```
 tar cf - xxx |  rsh host tar xfp -
```

and it worked?

5.4 Key Management

No discussion on IPsec could be complete without mentioning key management. In our case we assign keys for the IPsec tunnels statically. We only use symmetric encryption (DES or 3DES) so we just create a file with random numbers and the scripts use these to generate the secure associations.

This creates the problem of distribution of the floppies since they contain the keys. The alternative scenario of assigning each institution a private key and using a key management protocol to create the session keys was considered in the initial stages of the design but was deemed to add too much complexity to the system. In any case, if the number of hosts increases so much that key generation and distribution becomes a problem, we can revise our original decision.

5.5 What we'd do differently

This title is often a euphemism for what we did wrong. In retrospect we spent way too much time with the floppy distribution. That was a mistake because floppies are a relic of the past and there are already quite cost effective replacements offering capacities in excess of 100Mb. However running from a ramdisk is probably a good idea anyway, since it allows the use of read-only distribution media.

Another really big mistake was interfering with the OpenBSD startup files in /etc. Since we had a number of FreeBSD servers already running and we used PICOBSD (which is also based on FreeBSD), we decided to stick with the FreeBSD style startup files. Don't do it folks! There are some "minor" differences between the two systems that make porting these files a nightmare. Also, changing these files in a substantial way, creates a platform that cannot be easily upgraded to the next OpenBSD release since all this work will have to be redone. The worst is that, by that time, we'll have probably forgotten most of what we have learned in the first port so that recurring mistakes will bound to happen. All this without considering the wrath of the OpenBSD crowd. In retrospect, I would advise creating a list of the actions that will have to be performed on startup. This list can be drawn by examining the PICOBSD files. Then using this list we can modify the OpenBSD startup files creating diffs that can ease the next operating system upgrade.

6. Conclusions – Future Plans

Most of what the tools used in our project are well known and widely used. There are numerous network monitoring programs, and the notion of having a monitoring station installed in a LAN, has been proposed before [Ches94]. Also the use of IPsec, ssh and the other tools mentioned in this paper is common practice.

However, putting all this stuff on a floppy and deploying numerous stations both in our university network and in the institutions participating in the GUnet pilot is to our knowledge the first clear demonstration of such a network monitoring station.

It is difficult to stay still in an ever changing world, so we intend to keep maintaining the software distribution for the network monitoring station and adding features to make our lives easier. Taking advantage of the fact that the entire system is based on free software, we plan to make both bootable floppies and the development system that produces them available on our ftp site, so that other users may benefit from our efforts.

We are also looking into ways of abandoning the floppies as distribution media and replacing them with flash RAM cards because they offer higher capacity than floppies and, due to the complete lack of moving parts, are far more reliable.

Finally, as vendors slowly adopt the IPsec protocol, we hope to integrate equipment that support the protocol into our secure network. This will give us first hand experience with interoperability and integration between different implementations of the IPsec protocol.

References

[Chap95] Chapman D.Brent and Elizabeth D. Zwicky, "Building Internet Firewalls," Second Edition, O'Reilly & Associates, Inc. 1995.

[Ches94] Cheswick, William and Steven Bellovin, "Firewalls & Internet Security, Repelling the Wily Hacker," Addison-Wesley Professional Computing Series, 1994.

[Garf96] Garfinkel, Simpson and Gene Spafford, "Practical UNIX and Internet Security," Second Edition, O'Reilly & Associates, Inc. 1996.

[Reis93] Reiss, Levi and Joseph Radin, "Unix System Administration Guide," Osborne McGraw Hill Inc. 1993.

[Scot98] Scott, Charlie, Paul Wolfe and Mike Erwin, "Virtual Private Networks," O'Reilly & Associates, Inc. 1998.

[Shah97] Shah, Deval and Helen Holzbaur, "*Virtual Private Networks: Security With an Uncommon Touch,*" Data Communications, Sept. 97,

[Silv98] da Silva, James, "Cruchgen," OpenBSD User Manual, 1998.

[Stal95] Stallings, William, "Network and Internetwork Security, Principles and Practice" Prentice Hall, 1995

The Flask Security Architecture: System Support for Diverse Security Policies

Ray Spencer *Secure Computing Corporation*
Stephen Smalley, Peter Loscocco *National Security Agency*
Mike Hibler, David Andersen, Jay Lepreau *University of Utah*

http://www.cs.utah.edu/flux/flask/

Abstract

Operating systems must be flexible in their support for security policies, providing sufficient mechanisms for supporting the wide variety of real-world security policies. Such flexibility requires controlling the propagation of access rights, enforcing fine-grained access rights and supporting the revocation of previously granted access rights. Previous systems are lacking in at least one of these areas. In this paper we present an operating system security architecture that solves these problems. Control over propagation is provided by ensuring that the security policy is consulted for every security decision. This control is achieved without significant performance degradation through the use of a security decision caching mechanism that ensures a consistent view of policy decisions. Both fine-grained access rights and revocation support are provided by mechanisms that are directly integrated into the service-providing components of the system. The architecture is described through its prototype implementation in the Flask microkernel-based operating system, and the policy flexibility of the prototype is evaluated. We present initial evidence that the architecture's impact on both performance and code complexity is modest. Moreover, our architecture is applicable to many other types of operating systems and environments.

1 Introduction

A phenomenal growth in connectivity through the Internet has made computer security a paramount concern, but no single definition of security suffices. Different computing environments, and the applications that run in them, have different security requirements. Because any notion of security is captured in the expression of a security policy, there is a need for many different policies and even many types of policies [1, 43, 48]. To be generally acceptable, any computer security solution must be flexible enough to support this wide range of security policies. Even in the distributed environments of today, this policy flexibility must be supported by the security mechanisms of the operating system [32].

Supporting policy flexibility in the operating system is a hard problem that goes beyond just supporting multiple policies. The system must be capable of supporting fine-grained access controls on low-level objects used to perform higher-level functions controlled by the security policy. Additionally, the system must ensure that the propagation of access rights is in accordance with the security policy. Lastly, policies are not, in general, static. To cope with policy changes or dynamic policies, the system must have a mechanism for revoking previously granted access rights. Earlier systems have provided mechanisms that allow several security policies to be supported, but they are inadequate to generally support policy flexibility because they fail to address at least one of these three areas.

This paper describes an operating system security architecture that demonstrates the feasibility of policy flexibility. This is done by presenting its prototype implementation, the Flask microkernel-based operating system, that successfully overcomes these obstacles to policy flexibility. The cleaner separation of mechanism and policy specified in the security architecture enables a richer set of security policies to be supported with less policy-specific customization than has previously been possible. Flask includes a security policy server to make access control decisions and a framework in the microkernel and other object managers in the system to enforce those access control decisions. Although the prototype system is microkernel-based, the security mechanisms do not depend on a microkernel architecture and will easily generalize beyond it.

The resulting system provides policy flexibility. It supports a wide variety of policies. It controls the propagation of access rights by ensuring that the security policy is consulted for every access decision. Enforcement mechanisms, directly integrated into the service-providing components of the system, enable fine-grained

This research was supported in part by the Defense Advanced Research Projects Agency in conjunction with the Department of the Army under contract DABT63–94–C–0058 and with the Air Force Research Laboratory, Rome Research Site, USAF, under agreement F30602–96–2–0269. It was also supported in part by the Maryland Procurement Office, contract MDA904-97-C-3047.

Authors: {sds,pal}@epoch.ncsc.mil, {mike,danderse,lepreau}-@cs.utah.edu, saltalk@lakenet.com (Spencer).

access controls and dynamic policy support that allows the revocation of previously granted access rights. Initial performance results, as well as statistics on the scale and invasiveness of the code changes, indicate that the impact of policy flexible security on the system can be kept to a minimum.

The remainder of the paper begins by elaborating on the meaning of policy flexibility. After a discussion of why two popular mechanisms employed in systems to provide security are limiting to policy flexibility, some related work is described. The Flask architecture is then presented through a discussion of its prototype design and implementation. The paper concludes with an evaluation of the policy flexibility of the system, an assessment of the performance impact, and a discussion of the scale and invasiveness of the Flask changes.

2 Policy Flexibility

When first attempting to define security policy flexibility, it is tempting to generate a list of all known security policies and define flexibility through that list. This ensures that the definition will reflect a real-world view of the degree of flexibility. Unfortunately, this simplistic definition is unrealistic. Real-world security polices in computer systems are limited by the mechanisms currently provided in such systems, and it is not always clear how security policies enforced in the "pencil-and-paper" world translate to computer systems, if at all [3, 48]. As such, a better definition is needed.

It is more useful to define security policy flexibility by viewing a computer system abstractly as a state machine performing atomic operations to transition from one state to the next. Within such a model, a system could be considered to provide total security policy flexibility if the security policy can interpose atomically on any operation performed by the system, allowing the operation to proceed, denying the operation, or even injecting operations of its own. In such a system, the security policy can make its decisions using knowledge of the entire current system state, where the current system state can be considered to encompass the history of the system. Because it is possible to interpose on all access requests, it is possible to modify the existing security policy and to revoke any previously granted access.

This second definition more correctly captures the essence of policy flexibility, but practical considerations force a slightly more limited point of view. It is unlikely that a real system could base security policy decisions for all possible operations on the entire current system state. Instead, a more realistic approach is to identify that portion of the system state that is potentially security relevant and to control operations that affect or are affected by that portion of the state. The degree of flexibility in such a system will naturally depend upon the completeness of both the set of controlled operations and the portion of the current system state that is available to the security policy. Furthermore, the granularity of the controlled operations affects the degree of flexibility because it impacts the granularity at which sharing can be controlled.

This description of policy flexibility seems limiting in three ways. It allows some operations to proceed outside of the control of the security policy, restricts the operations that may be injected by the security policy, and permits some system state to exist beyond the scope of the security policy. In actuality, each of these apparent limitations is a desirable property since many of the internal operations and state of any system are of no apparent use or concern to any security policy. Section 6.1 will discuss how these limitations were interpreted for the Flask system.

A system that is policy flexible must be capable of supporting a wide variety of security policies. Security policies may be classified according to certain characteristics, including such things as: the need to revoke previously granted accesses, the type of input required to make access decisions, the sensitivity of policy decisions to external factors like history or environment, and the transitivity of access decisions [43, Sec. 6]. The remainder of this section focuses on revocation, which is the most difficult of these characteristics to support.

Since even the simplest security policies undergo change (*e.g.*, as user authorizations change), a policy flexible system must be capable of supporting policy changes. Since policy changes may be interleaved with the execution of controlled operations, there is the risk that the system will enforce access rights according to an obsolete policy. Thus, there must be effective atomicity in the interleaving of policy changes and controlled operations.

The fundamental difficulty in achieving this atomicity is ensuring that previously granted permissions can be revoked as required by a policy change. When a permission is to be revoked, the system must ensure that any service controlled by the permission will no longer be provided unless the permission is later granted again. Revocation can be a very difficult property to satisfy because permissions, once granted, have a tendency to migrate throughout the system. The revocation mechanism must guarantee that all of these migrated permissions are indeed revoked.

A basic example of a migrated permission surfaces in Unix. The access decision for writing to a file is performed when that file is opened, and the granted permission is cached in the file description for efficient validation of write access during write operations. Revoking write access to that file in Unix only prevents future attempts to open the file with write access and has no effect on the migrated permissions in existing file descriptions. This revocation support may be insufficient to meet the needs of a security policy. This type of situation is not uncommon, and migrated permissions can be found in other places throughout a system including: capabilities, access rights in page tables, open IPC connections, and operations currently in progress. More complicated systems are likely to yield more places to which permissions can migrate.

In most cases, revocation can be accomplished simply by altering a data structure. However, it is more complicated to revoke a permission when there is an operation in progress that has checked the permission already. The revocation mechanism must be able to identify all in-progress operations affected by such revocation requests and deal with each of them in one of three possible ways. The first is to abort the in-progress operation, returning an error status. Alternately, it could be restarted, allowing another check for the retracted permission. The third option is just to wait for the operation to complete on its own. In general, only the first two are safe. Only when the system can guarantee that the operation can complete without causing the revocation request to block indefinitely (*e.g.*, if all appropriate data structures have already been locked and there are no external dependencies) may the third option be taken. This is critical because blocking the revocation effectively denies the revocation request and causes a security violation.

3 Insufficiency of Popular Mechanisms

This section discusses two popular mechanisms that are often employed to provide security to systems and the reasons why both are limiting to policy flexibility in normal usage. However, each has benefits despite its limitations, and both can be used within Flask in restricted ways that allow some of their benefits without incurring their limitations.

3.1 Capability-Based Systems

The goal of a single operating system mechanism capable of supporting a wide range of security policies is not a new goal. The Hydra operating system developed in the 1970's separated its access control mechanisms from the definition of its security policy [29, 52]. Hydra was a capability-based system, although the developers of the system recognized the limitations of a simple capability model and introduced several enhancements to the basic capability mechanisms. The Hydra approach was taken even further by the KeyKOS [40] and EROS [47] systems. Though popular, capability mechanisms are poorly suited to providing policy flexibility, because they allow the holder of a capability to control the direct propagation of that capability, whereas a critical requirement for supporting security policies is the ability to control the propagation of access rights in accordance with the policy. The enhancements introduced by Hydra and KeyKOS are intended to limit such propagation, but the resulting systems still generally only support the specific policies they were designed to satisfy, at the cost of significant complexity that diminishes the attraction of the capability model in the first place.

Primarily with an interest in solving the problem of supporting a multilevel security policy within a capability-based system, a few capability-based systems (*e.g.*, SCAP [25], ICAP [18], Trusted Mach [4]) introduced mechanisms that validated every propagation or use of a capability against the security policy. Kain and Landwehr [23] developed a taxonomy to characterize such systems. In these systems, the simplicity of the capability mechanism is retained, but capabilities serve only as a least privilege mechanism rather than a mechanism for recording and propagating the security policy. This is a potentially valuable use of capabilities. However, the designs for these systems do not define the mechanisms by which the security policy is queried to validate capabilities, and those mechanisms are essential to providing policy flexibility. The Flask architecture described in this paper could be employed to provide the security decisions needed to validate the capabilities in these systems. In the Flask prototype, the architecture is used in exactly this way.

3.2 Intercepting Requests

A common approach used to add security to a system is to intercept service requests or to otherwise interpose a layer of security code between all applications and the operating system (*e.g.*, Kernel Hypervisors [37], SPIN [20]), or between particular applications or sets of applications (*e.g.*, L3/L4 [30], Lava [22], KeySAFE [28]). This may be done in capability systems or non-capability systems, and when applied to an operating system the security layer may lie within the operating system itself (as in Spring [36]) or in a component outside of the operating system to which all requests are redirected (as in Janus [17]).

However, this approach has some serious limitations. In order to add security by intercepting requests, the existing functional interface must expose all abstractions and information flows that the security policy wishes to control. To avoid maintaining redundant state in the access control layer, the functional interface must ensure that all security-relevant attributes are either directly available as parameters or easily derived from parameters. A policy that requires the use of some internal state of the object manager as an input to the decision can not be implemented without either changing the manager to export the state or, if possible, replicating the state management in the enforcer itself. The level of abstraction provided by the interface may be inappropriate or may cause difficulties in guaranteeing uniqueness or atomicity. For example, typical name-based calls suffer from issues of aliasing, multi-component lookups, and preserving the tranquility of the name-to-object mapping from the time-of-check to the time-of-use. Finally, this approach is limited in that the security layer can only affect the operation of the system as requests pass through it. Hence, it is often impossible for the system to reflect subsequent changes to the security policy, in particular, the revocation of migrated permissions.

As was the case with capabilities, implementing access control within a security layer is a good approach when these disadvantages can be avoided through the use of other mechanisms. However, it is important to recognize that other mechanisms are necessary, often mechanisms that are more invasive than intercepting requests, in order to provide any degree of flexibility in supporting security policies.

4 Related Work

The previous section described the relationship between Flask and a variety of efforts that involved capability-based systems or the interception of requests. This section describes the relationship between Flask and other efforts not previously mentioned. We focus on the research most directly related to Flask, although there are many other efforts with some relation to our work.

The security architecture of the Flask system is derived from the architecture of our previous prototype system DTOS [35], which had similar goals. However, while the DTOS security mechanisms were independent of any particular security policy, the mechanisms were not sufficiently rich to support some policies [43], especially dynamic security policies.

At the highest level of abstraction, the flexible security model for Flask is consistent with the Generalized Framework for Access Control (GFAC) [2]. However, the GFAC model assumes that all controlled operations in the system are performed in the same atomic operation in which the policy is consulted, which is very difficult to achieve in a practical system and is the primary obstacle that the Flask system has had to overcome.

The specific issue of revocation is not a new issue in operating system design, although it has received surprisingly little recognition. Multics [39] effectively provided immediate revocation of all memory permissions by invalidating segment descriptors. Redell and Fabry [42], Karger [24] and Gong [18] all describe approaches for revoking previously granted capabilities, though none were actually implemented. Spring [49] implemented a capability revocation technique, though only the capabilities were revoked, not migrated permissions. Revocation of memory permissions is naturally provided by microkernel-based systems with external paging support, such as Mach [31], though revocation is not extended to other permissions. DTOS provided the security server with the ability to remove permissions previously granted and stored in the microkernel's permission cache. However, except for memory permissions where Mach's mechanisms could be used, DTOS did not provide for revocation of migrated permissions [38].

The Flask prototype is implemented within a microkernel-based operating system with hardware-enforced address space separation between processes. Several recent efforts (*e.g.*, SPIN [5], VINO [46] and the Java protection models in [50]) have presented software-enforced process separation. The distinction is essentially irrelevant for the Flask architecture. It is essential that some form of separation between processes be provided, but the particular mechanism is not mandated by the Flask architecture. The general applicability of key aspects of the Flask architecture to other systems was concretely demonstrated by the adoption of the DTOS architecture in the security framework of SPIN [20]. Indeed, we believe the abstract Flask architecture, and the lessons it teaches, can be applied to software other than operating systems, such as middleware or distributed systems, although of course vulnerability to insecurities in the underlying operating systems would remain.

5 Flask Design and Implementation

This section defines the components of the Flask security architecture and identifies the requirements on each component necessary to meet the goals of the system. The Flask security architecture is described here in the context of its implementation within a microkernel-based multiserver operating system. However, the security architecture only requires that the operating system include a reference monitor [16, Ch. 10]. In particular, the architecture requires the completeness and isolation properties, although verifiability is also ultimately necessary for confidence in any implementation of the architecture.

The Flask prototype was derived from the Fluke microkernel-based operating system [14]. The Fluke microkernel is especially well-suited for implementing the Flask architecture due to its lack of global resources [14] and the atomic properties of its API [13]. However, the original Fluke system was capability-based and was not in itself adequate to meet the requirements of the Flask architecture.

The remainder of this section starts by providing an

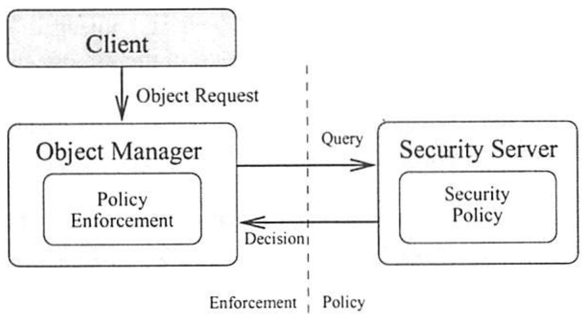


Figure 1: The Flask architecture. Components which enforce security policy decisions are referred to as *object managers*. Components which provide security decisions to the object managers are referred to as *security servers*. The decision making subsystem may include other components such as administrative interfaces and policy databases, but the interfaces among these components are policy-dependent and are therefore not addressed by the architecture.

overview of the Flask architecture. Then, it describes general support mechanisms required for the basic Flask architecture. It discusses the specific changes required for the microkernel. It explains how the complications caused by the need for revocation were overcome. This section ends by describing the prototype security server.

5.1 Architecture Overview

The Flask security architecture [44], as shown in Figure 1, describes the interactions between subsystems that enforce security policy decisions and a subsystem which makes those decisions, and the requirements on the components within each subsystem. The primary goal of the architecture is to provide for flexibility in the security policy by ensuring that these subsystems always have a consistent view of policy decisions regardless of how those decisions are made or how they may change over time. Secondary goals for the architecture include application transparency, defense-in-depth, ease of assurance, and minimal performance impact.

The Flask security architecture provides three primary elements for object managers. First, the architecture provides interfaces for retrieving access, labeling and polyinstantiation decisions from a security server. Access decisions specify whether a particular permission is granted between two entities, typically between a subject and an object. Labeling decisions specify the security attributes to be assigned to an object. Polyinstantiation decisions specify which member of a polyinstantiated set of resources should be accessed for a particular request. Second, the architecture provides an *access vector cache* (AVC) module that allows the object manager to cache access decisions to minimize the performance overhead. Third, the architecture provides object managers the ability to register to receive notifications of changes to the security policy.

Object managers are responsible for defining a mechanism for assigning labels to their objects. A control policy, which specifies how security decisions are used to control the services provided by the object manager, must be defined and implemented by each object manager. This control policy addresses threats in the most general fashion by providing the security policy with control over all services provided by the object manager and by permitting these controls to be configurable based on threat. Each object manager must define handling routines which are called in response to policy changes. For all uses of polyinstantiation, each object manager must define the mechanism by which the proper instantiation of a resource is chosen.

5.2 General Support Mechanisms

This section describes general support mechanisms that were introduced for all of the object managers in order to support policy flexibility. Despite the simplicity of the Flask architecture, some subtleties arise in the implementation, as will be discussed below.

5.2.1 Object Labeling All objects that are controlled by the security policy are also labeled by the security policy with a set of security attributes, referred to as a security context. A fundamental issue in the architecture is how the association between objects and security contexts is maintained. The simplest solution would be to define a single policy-independent data type which is part of the data associated with each object. However, no single data type is well-suited to all of the differing ways in which labels are used in a system. The Flask architecture addresses these conflicting needs by providing two policy-independent data types for labeling.

A *security context*, the first policy-independent data type, is a variable-length string which can be interpreted by any application or user with an understanding of the security policy. A security context might consist of several attributes, such as a user identity, a classification level, a role and a type enforcement [6] domain, but this depends on the particular security policy. As long as it is treated as an opaque string, a security context can be handled by an object manager without compromising the policy flexibility of the object manager. However, using security contexts for labeling and policy decision lookups would be inefficient and would increase the likelihood of policy-specific logic being introduced into the object managers.

The second policy-independent data type, the *security identifier* (SID), is defined by Flask to be a fixed-size value which can be interpreted only by the security server and is mapped by the security server to a particu-

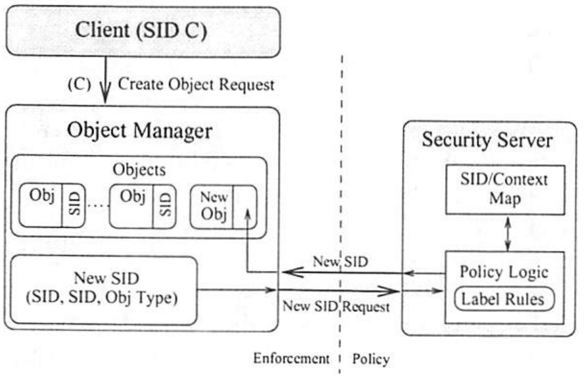


Figure 2: Object labeling in Flask. A client requests the creation of a new object from an object manager, and the microkernel supplies the object manager with the SID of the client. The object manager sends a request for a SID for the new object to the security server, with the SID of the client, the SID of a related object and the object type as parameters. The security server consults the labeling rules in the policy logic, determines a security context for the new object, and returns a SID that corresponds to that security context. Finally, the object manager binds the returned SID to the new object.

lar security context. Possession or knowledge of a SID for a given security context does not grant any authorization for that security context. The SID mapping cannot be assumed to be consistent across executions (reboots) of the security server nor across security servers on different nodes. Consequently, SIDs may be lightweight; in the implementation, SIDs are simply 32-bit integers. There is no specified internal structure to a SID; any internal structure is known only by the security server. The SID allows most object manager interactions to be independent of not just the content but even the format of a security context, simplifying object labeling and the interfaces that coordinate the security policy between the security server and object managers. However, in some cases, such as labeling persistent objects or labeling objects which are exported to other nodes, object managers must handle security contexts. This is described further in the discussion of the file server and network server in Section A.1 and Section A.2.

When an object is created, it is assigned a SID that represents the security context in which the object is created. This context typically depends upon the client requesting the object creation and upon the environment in which it is created. For example, the security context of a newly created file is dependent upon the security context of the directory in which it is created and the security context of the client that requested its creation. Since the computation of a security context for a new or transformed object may involve policy-specific logic, it cannot be performed by the object manager itself. The labeling of a new object is depicted in Figure 2. For some security policies, such as an ORCON policy [19, 34], the security policy may need to uniquely distinguish subjects and objects of certain classes even if they are created in the same security context. For such policies, the SID must be computed from the security context and a unique identifier chosen by the security server.

5.2.2 Client and Server Identification

Object managers must be able to identify the SID of a client making a request when this SID is part of a security decision. It is also useful for clients to be able to identify the SID of a server to ensure that a service is requested from an appropriate server. Hence, the Flask architecture requires that the underlying system provide some form of client and server identification for inter-process communication (IPC). However, this feature is not complete without providing the client and server a means of overriding their identification. For instance, the need of a subject to limit its privileges when making a request on behalf of another subject is one justification for capability-based mechanisms [21]. In addition to limiting privileges, overriding the actual identification can be used to provide anonymity in communications or to allow for transparent interposition, such as through a network IPC server connecting the client and server in a distributed system [11].

The Flask microkernel provides this service directly as part of IPC processing, rather than relying upon complicated and potentially expensive external authentication protocols such as those in Spring and the Hurd [7]. The microkernel provides the SID of the client to the server along with the client's request. The client can identify the SID of the server by making a kernel call on the capability to be used for communication. When making an IPC request, the client can specify a different SID as its effective SID to override its identification to the server. The server can also specify an effective SID when preparing to receive requests. In both cases, permission to specify a particular effective SID is decided by the security server and enforced by the microkernel. Thus, the Flask microkernel supports the basic access control and labeling operations required for the architecture and it provides the flexibility needed for least privilege, anonymity or transparent interposition.

5.2.3 Requesting and Caching Security Decisions

In the simplest implementation, the object manager can make a request to the security server every time a security decision is needed. However, to alleviate the performance impact of communicating with the security server for each decision and of the computation of the decision within the security server, the Flask architecture provides caching of security decisions within the object manager.

The caching mechanisms in Flask provide much more than simply caching individual security decisions. The access vector cache (AVC) module, which is a common

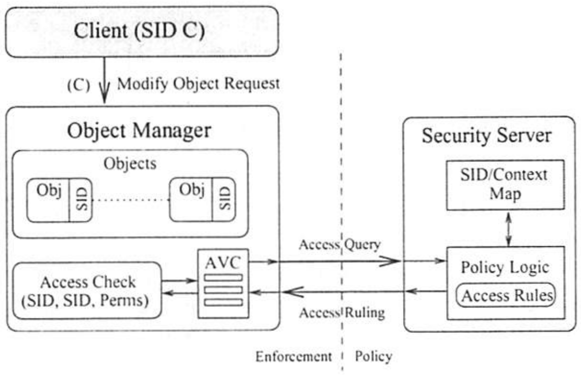


Figure 3: Requesting and caching security decisions in Flask. A client requests the modification of an existing object from an object manager. The object manager queries its access vector cache (AVC) module for an access ruling for the (client SID, object SID, requested permissions) triple. If no valid entry exists, then the AVC module sends an access query to the security server. The security server consults the access rules in the policy logic, determines an access ruling, and returns the access ruling to the AVC module.

library shared by the object managers, provides for the coordination of the policy between the object manager and the security server. This coordination addresses both requests from the object manager for policy decisions and requests from the security server for policy changes. The first of these is discussed in this section, while the second is discussed in Section 5.4.

For a typical controlled operation in Flask, an object manager must determine whether a subject is allowed to access a object with some permission or set of permissions. The sequence of requesting and caching security decisions is depicted in Figure 3. To minimize the overhead of security computations and requests, the security server can provide more decisions than requested, and the AVC module will store these decisions for future use. When a request for a security decision is received by the security server, it will return the current state of the security policy for a set of permissions with an *access vector*. An access vector is a collection of related permissions for the pair of SIDs provided to the security server. For instance, all file access permissions are grouped into a single access vector.

5.2.4 Polyinstantiation Support A security policy may need to restrict the sharing of a fixed resource among clients by polyinstantiating the resource and partitioning the clients into sets which can share the same instantiation of the resource. For example, multi-level secure Unix systems frequently partition the /tmp directory, maintaining separate subdirectories for each security level [51]; the corresponding solution for Flask is discussed in Section A.1. A similar issue arises with the

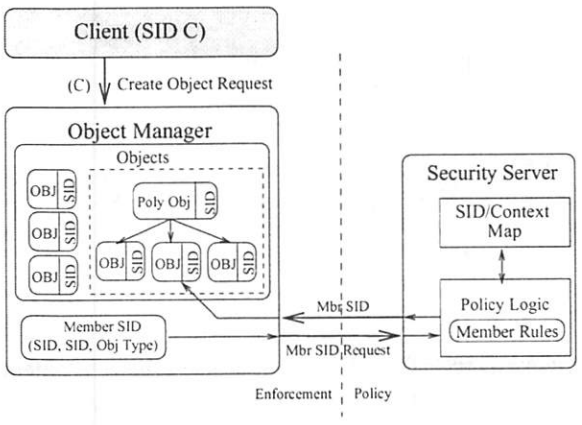


Figure 4: Polyinstantiation in Flask. A client requests the creation of a new object from an object manager, and the microkernel supplies the object manager with the SID of the client. The object manager sends a request for a SID for the member object to the security server, with the SID of the client, the SID of the polyinstantiated object and the object type as parameters. The security server consults the polyinstantiation rules in the policy logic, determines a security context for the member, and returns a SID that corresponds to that security context. Finally, the object manager selects a member based on the returned SID, and creates the object as a child of the member.

TCP or UDP port spaces, as discussed in Section A.2. The Flask architecture supports polyinstantiation by providing an interface by which the security server may identify which instantiation can be accessed by a particular client. Both the client and the instance are identified by SIDs. The instantiations are referred to as *members*. The general sequence of selecting a member is depicted in Figure 4.

5.3 Microkernel-specific Features

The previous sections described the security functions that are common to all of the Flask object managers. In this section, we discuss the specific features that have been added to the microkernel. Support for revocation, however, will be discussed separately in Section 5.4. The specific features that were added to some of the other Flask object managers are described in Appendix A.

Due to the requirements of Fluke's architecture, each active kernel object is associated with a small chunk of physical memory [14]. Though "memory" is not itself an object within the microkernel, the microkernel provides the base service for memory management and binds a SID to each memory segment. The SID of each kernel object is identical to the SID of the memory segment with which it is associated. This relationship between the label of memory and the label of kernel objects associated with that memory permits the Flask microkernel controls to leverage the existing protection model of Fluke, rather than introducing an orthogonal protection

SOURCE	TARGET	PERMISSION
Client SID	Effective Client SID	SpecifyClient
Server SID	Effective Server SID	SpecifyServer
Effective Client SID	Effective Server SID	Connect

Table 1: Permission requirements for an IPC connection to exist. The specify permissions are only required when a subject specifies an effective SID. If a subject does not specify an effective SID, then its effective SID is equal to its actual SID.

model as in DTOS. However, it also creates a potential loss of labeling flexibility, since the memory allocation granularity is much coarser than the allocation granularity for kernel objects.

Flask provides direct security policy control over the propagation of memory access modes by associating a Flask permission with each mode, based on the SID of the address space and the SID of the memory segment. These memory access modes also act as capabilities to kernel objects associated with the memory. During the initial attempt to access mapped memory, the microkernel verifies that the security policy explicitly grants permission for each requested access mode. Memory permissions cannot be computed at the level of any interface in Fluke, and are computed instead during page faults; hence, these controls provide an example where merely intercepting requests would be insufficient. Since the SID of a memory segment is not allowed to change, the Flask permissions need only be revalidated if a policy change occurs, as discussed in Section 5.4.

In Fluke, a port reference serves as a capability for performing an IPC to a server thread waiting on the corresponding port set. Control over propagation in Fluke may be performed through typical interposition techniques. In contrast, Flask provides direct control over the use of such port references by only allowing an IPC connection between two subjects if the appropriate permissions shown in Table 1 are satisfied. These direct controls permit the policy to regulate the use of capabilities, addressing the concerns of Section 3.1.

An interesting aspect of the Flask microkernel is the controls that are imposed on relationships between objects. In Fluke, these relationships are defined through the use of object references (*e.g.* the state of a thread contains an address space reference). Unfortunately, these references are used in many different ways, in contrast to the way in which read and write access modes are used to control access to kernel objects. For example, a reference to an address space may be used to map memory into the space or to export memory from the space. Hence, Flask introduces separate controls over these relationships and provides finer-grained control than Fluke. Some of the controls simply require the two objects to have equal SIDs, while others involve explicit permissions, as described in detail in [44, Sec. 3].

5.4 Revocation Support Mechanisms

The most difficult complication in the Flask architecture is that the object managers effectively keep a local copy of certain security decisions, both explicitly in an access vector cache and implicitly in the form of migrated permissions. Therefore a change to the security policy requires coordination between the security server and the object managers to ensure that their representations of the policy are consistent. This section is devoted to a more detailed discussion of the requirements on the components of the architecture during a change in security policy.

The need for effective atomicity stated in Section 2 is achieved by imposing two requirements on the system. The first is that after completion of a policy change, the behavior of the object manager must reflect that change. No further controlled operations requiring a revoked permission can be performed without a subsequent policy change. The second requirement is that object managers must complete policy changes in a timely manner.

This first requirement is only a requirement on the object managers, but it results in effective atomicity of system-wide policy when coupled with a well-defined protocol between the security server and the object managers. This protocol involves three steps. First, the security server notifies all object managers that may have been previously provided any portion of the policy that has changed. Second, each object manager updates its internal state to reflect the change. Finally, each object manager notifies the security server that the change is complete. Sequence numbers are used to address the interleaving of messages providing policy decisions to the object managers and messages requesting changes to the policy. Both the synchronization protocol, which has been implemented, and an alternative approach based on theories of database consistency are described in [45, Sec. 6]. The latter solution was drawn from a model of transactional consistency, but solutions related to distributed shared memory consistency may also serve as useful models.

The last step of the protocol is essential to support policies that require policy changes to occur in a particular order. For instance, a policy may require that certain permissions be revoked prior to granting new permissions. The security server cannot consider a policy change to be completed until it is completed by all affected object managers. This allows effective atomicity of system-wide policy changes since the security server can determine when the policy change is effective for all relevant object managers.

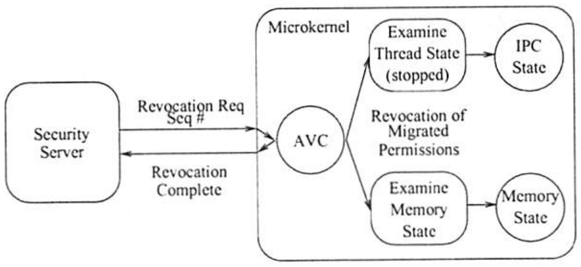


Figure 5: A revocation of microkernel permissions. Upon receipt of a revocation request from the security server, the microkernel first updates its access vector cache, and then proceeds to examine thread and memory state and perform revocations as necessary. The atomic properties of Fluke were leveraged to ease implementation of the revocation mechanism.

This protocol does not impose an undue burden in state management on the security server. The number of object managers in many systems is relatively small and the only transactions which require additional state are those where an object manager initially issues an access query for a permission that is granted. Furthermore, the security server may track permission grantings at various granularities to reduce the amount of state recorded by the security server.

The form of atomicity provided by the protocol is reasonable because of the timeliness requirement imposed on the object managers. It must not be possible for the revocation request to be arbitrarily delayed by actions of untrusted software. Each object manager must be capable of updating its own state without being indefinitely blocked by its clients. When this timeliness requirement is generalized for system-wide policy changes, it also involves two other elements of the system: the microkernel, which must provide timely communication between the security server and object managers, and the scheduler, which must provide the object manager with CPU resources.

The general AVC module handles the initial processing of all policy change requests and updates the cache appropriately. The only other operation that must be performed is revocation of migrated permissions. After updating the cache, the AVC module invokes any callbacks which have been registered by the object manager for revoking migrated permissions. The file server supports revocation of permissions which have migrated into file description objects, but currently lacks support for interrupting in-progress operations. Complete callbacks for revoking migrated permissions have currently been implemented only within the Flask microkernel, as shown in Figure 5.

Two properties of the Fluke API simplify revocation in the microkernel: it provides prompt and complete exportability of thread state and guarantees that all kernel operations are either atomic or cleanly subdivided into user-visible atomic stages [13]. The first property permits the kernel revocation mechanism to assess the kernel's state, including operations currently in progress. The revocation mechanism may safely wait for operations currently in progress to complete or restart due to the promptness guarantee. The second property permits Flask permission checks to be encapsulated in the same atomic operation as the service that they control, thereby avoiding any occurrences of the service after a revocation request has completed.

5.5 The Security Server

As stated earlier, the security server is required to provide security policy decisions, to maintain the mapping between SIDs and security contexts, to provide SIDs for newly created objects, to provide SIDs of member objects, and to manage object manager access vector caches. Additionally, most security policy server implementations will provide functionality for loading and changing policies. A security server might also benefit from providing its own caching mechanism, in addition to those contained in the object managers, to hold the results of access computations. This may prove advantageous because the security server can improve its response time by using cached results from previous, potentially expensive, access computations requested by any client.

The security server also is typically a policy enforcer over its own services. First of all, if the security server provides interfaces for changing the policy, it must enforce the policy over which subjects can access this interface. Second, it may limit the subjects that can request policy information. This is especially important in a policy where permission requests alter the policy, such as a dynamic conflict of interest policy. If the confidentiality of the policy information is important, then object managers that cache policy information must also be responsible for its protection.

In a distributed or networked environment, it is tempting to suggest that the security server of each node merely act as a local cache of the environment's policy. However, to support heterogeneous policy environments, it is desirable for each node to have its own security server with a locally defined policy component, with some degree of coordination at a higher level. Even in a homogeneous policy environment, a core portion of the security policy must be locally defined for the node in order to securely bootstrap the system into a state where it may consult the environment's policy. The development of a distributed security server for coordinating the per-node security servers within an environment remains as future work. For many policies, the security server should easily be scalable and replicable, since most policies will require little interaction among the individual nodes' security servers. However, some security policies, such as history-based policies, may require greater coordination among the security servers.

The security policy encapsulated by the Flask security server is defined through a combination of its code and a policy database. Any security policy that can be expressed through the prototype's policy database language may be implemented simply by altering the policy database. Supporting additional security policies requires changes to the security server's internal policy framework through code changes or by completely replacing the security server. It is important to note that even security policies that require altering the code of the security server do not require any changes to the object managers.

The current Flask security server prototype implements a security policy that is a combination of four subpolicies: multi-level security (MLS) [3], type enforcement [6], identity-based access control and dynamic role-based access control (RBAC) [10]. The access decisions provided by the security server must meet the requirements of each of these four subpolicies. The policy logic for the multi-level security policy is largely defined through the security server code, aside from the labels themselves. The policy logic for the other subpolicies is primarily defined through the policy database language. These four subpolicies are not all the policies supported by the architecture or its implementation in Flask. They were chosen for implementation in the security server prototype in order to exercise the major features of the architecture.

Because the Flask effort has focused on policy enforcement mechanisms and the coordination between these mechanisms and the security policy, the set of additional security policies that can be implemented solely through changes to this policy database is currently limited. This is simply a shortcoming of the current prototype rather than a characteristic of the architecture. We have yet to explore the development of a more expressive policy specification language or policy configuration tool for Flask. Such a tool would facilitate the definition of new security policies in the current prototype. There have been several recent projects that do consider flexible tools for configuring the security policies (e.g., Adage [53], ASP [8], Dynamic DTE [15], ARBAC [41]) that nicely complement the Flask effort by potentially providing ways to manage the mechanisms provided by Flask.

6 Results

This section describes the results of the effort in three areas: policy flexibility, performance impact, and the scale and invasiveness of the code changes.

6.1 Flexibility in the Flask Implementation

We evaluate the policy flexibility that the system provides based upon the description of policy flexibility in Section 2. The most important criterion discussed in that section was "atomicity," i.e., the ability of the system to ensure that all operations in the system are controlled with respect to the current security policy. Section 5.4 described how the Flask architecture provides an effective atomicity for policy changes and how the microkernel in particular achieves atomicity for policy changes relating to its objects. Achieving this atomicity for the other object managers remains to be done.

Section 2 also identifies three other potential weaknesses in policy flexibility. The first is the range of operations that the system can control. As described in Section 5.3 and Appendix A, each Flask object manager defines permissions for all services which observe or modify the state of its objects and provides fine-grained distinctions among its services. The advantages of the Flask controls over merely intercepting requests were clearly illustrated.

The second potential source of inflexibility is the limitation on the operations that may be invoked by the security policy. In Flask, the security server may use any of the interfaces provided by the object managers. Furthermore, the Flask architecture provides the security server with the additional interfaces provided by the AVC module in each object manager. However, this is obviously not the same as having access to any arbitrary operation. For example, if the security policy requires the ability to invoke an operation which is strictly internal to some object manager, the object manager would have to be changed to support that policy.

The third potential source of inflexibility is the amount of state information available to the security policy for making security decisions. Based upon our previous analysis of policies for DTOS, the provision of a pair of SIDs is sufficient for most policies [43, Sec. 6.3]. However, the limitation to two SIDs is a potential weakness in the current Flask design. The description of the Flask file server in Section A.1 identifies one case where a permission ultimately depends upon three SIDs and must be reduced to a collection of permissions among pairs of SIDs. An even worse situation is if the security decision should depend upon a parameter to a request that is not represented as a SID. Consider a request to change the scheduling priority of a thread. Here the security policy must certainly be able to make a decision based in part on the requested priority. This parameter can be considered within the current implementation by defining separate permissions for some classes of changes, for instance, increasing the priority can be a different permission than decreasing the priority. But it is not practical to define a separate permission for every possible change to the priority.

This is not a weakness in the architecture itself, and the design could easily be changed to allow for a security decision to be represented as a function of arbitrary parameters. However, the performance of the system would certainly be impacted by such a change, because an access vector cache supporting arbitrary parameters would be much more complicated than the current cache. A better solution may be to expand the interface only for those specific operations that require decisions based upon more complex parameters, and to provide separate caching mechanisms for those decisions. The Flask prototype provides a research platform for exploring the need for a richer interface to better support policy flexibility.

6.2 Performance

All measurements in this section were taken using the time-stamp counter register on a 200MHz Pentium Pro processor with a 256KB L2 cache and 64MB of RAM. While a complete assessment of performance requires analysis of all object managers, we limit ourselves to the microkernel, and primarily to IPC since it is a critical path which must be factored into all higher level measurements.

6.2.1 Object Labeling The segment SID for any piece of mapped physical memory is readily available, since it is computed when a virtual-to-physical address translation is created and is stored along with that translation. As the address translation must be obtained at object creation time anyway, the additional cost of labeling is minimal. We verified this by measuring the cost to create the simplest kernel object in both Fluke and Flask, showing the worst case overhead. Flask added 1% to the operation (3.62 versus 3.66 μs).

6.2.2 IPC Operations This section presents performance measurements for IPC operations under various message sizes and also measures the impact of caching within the microkernel. Table 2 presents timings for a variety of client-server IPC microbenchmarks for the base Fluke microkernel and under different scenarios in the Flask system. The tests measure cross-domain transfer of varying amounts of data, from client to server and back again.

For all of the tests performed on Flask in Table 2, the required permissions are available in the access vector cache at the location identified by a "hint" within the port reference structure. While we have provided the data structures to allow for fast queries of previously computed security decisions, we have not done any specific code optimization to speed up the execution. Therefore it was encouraging to find that the addition of these data structures alone is sufficient to almost completely eliminate any measurable impact of the permission checks.

		Flask		
message size	Fluke (μs)	naive	client identification	client impersonation
``Null''	13.5	+2%	+9%	+6%
16-byte	15.0	+2%	+4%	+6%
128-byte	15.8	+1%	+2%	+5%
1k-byte	21.9	+2%	+2%	+4%
4k-byte	42.9	+1%	+1%	+2%
8k-byte	78.5	+1%	+5%	+1%
64k-byte	503	+0%	+6%	+0%

Table 2: Performance of IPC in Flask relative to the base Fluke system. A "Null" IPC actually transfers a minimal message, 8 bytes in the current implementation. In *Fluke*, the tests use the standard Fluke IPC interfaces in a system configured with no Flask enforcement mechanisms. Absolute times are shown in this column as a basis for comparison. *Naive* runs the same tests on the Flask microkernel. In *client identification*, the tests have been modified to use the Flask-specific server-side IPC interface to obtain the SID of the client on every call. *Client impersonation* uses the client-side IPC interface to specify an effective SID for every call.

The most interesting case in Table 2 is the *naive* column, because it represents the most common form of IPC in the Flask system. Along this path there is only a single *Connect* permission check. The results show a worst-case 2% (~50 machine cycle) performance hit. As would be expected, the relative effect of the single access check diminishes as the size of the data transfer increases and memory copy costs become the dominating factor. The *client identification* column has a larger than expected impact due to the fact that, in the current implementation, the client SID is passed across the interface to the server in a register normally used for data transfer. This forces an extra memory copy (particularly obvious in the Null IPC test). The significant effect on large data transfers is unexpected and needs to be investigated. The *client impersonation* column shows the impact of checking both the *Connect* and *SpecifyClient* permissions.

The effect of not finding the permission through the hint is shown in Table 3, which presents the relative costs of retrieving a security decision from the cache and from the security server. The operation being performed is the most sensitive of the IPC operations, round trip of transfer of a "null" message between a client and a server and is consequently representative of the worst case.

The *cache* column shows that the use of the hint is significant in that it reduces the overhead from 7% to 2%.

		Flask			
	Fluke	using hint	using cache	calling trivSS	calling realSS
`''Null''`	13.5 μs	13.8 μs +2%	14.4 μs +7%	43.4 μs +221%	82.5 μs +511%

Table 3: Marginal cost of security decisions in Flask. The first two columns repeat data from Table 2, identifying the relative cost of Flask when the required permission is found in the access vector cache (AVC) using the hint. The third column is the time required when the hint was incorrect but the permission was still found in the AVC. The *trivSS* column is the time required when the permission is not found in the AVC, and a "trivial" security server, which immediately returns an access ruling with all permissions granted, is used. The *realSS* column is the time required when the permission is not found in the AVC and an access ruling is computed by our prototype security server.

The *trivSS* column shows a more than tripling of the time required in the base Fluke case. The IPC interaction between the microkernel and security server requires transfer of 20 bytes of data to the security server (along with the client SID) and return of 20 bytes. Since the permission for this IPC interaction is found using the hint, we see from Table 2 that over half of the additional overhead is due to the IPC. The remainder of the overhead is due to the identification of the request for a security decision, construction of the security server request in the kernel, and the unmarshaling and marshaling of parameters in the security server itself. The additional overhead in the *realSS* column compared to the previous case is the time required to compute a security decision within our prototype security server. Though no attempt has been made to optimize the security server computations, this result points out that the access vector cache can potentially be important regardless of whether interactions with the security server require an IPC interaction.

6.2.3 Revocation Operations The possible microkernel revocation operations are described in Section 5.4. For demonstration purposes we chose to evaluate the most expensive of those operations, IPC revocation. Table 4 shows the results with varying numbers of active connections. The large base case is due to the need to stop all threads in the system when an IPC revocation is processed. The Fluke kernel provides a mechanism to cancel a thread and wait for it to enter a stopped state when the kernel wishes to examine or modify the thread's state. The stop operation cannot be blocked indefinitely by the thread's activities nor by the activities of any other thread. Since a thread must be stopped prior to examination in order to ensure that it is in a well-defined state, the current Flask implementation must stop all threads when an IPC revocation is processed. Thus, the current implementation meets the completeness and timeliness requirements of the architecture but is quite costly. In contrast, the actual cost to examine and update the state of the affected threads is small in relation, and as expected scales linearly with the number of connections. Changing the Fluke kernel to permit greater concurrency during the processing of a revocation request remains as future work.

connections	revocation time
1	1.55 ms
2	1.56 ms
4	1.57 ms
8	1.60 ms
16	1.65 ms

Table 4: Measured cost of revoking IPC connections. A connection is established from a client to a server and then is immediately revoked. Increasing numbers of interposed threads are used to increase the work done for each revocation.

The frequency of policy changes is obviously policy dependent, but the usual examples of policy changes are externally driven and therefore will be infrequent. Moreover, a performance loss in a system with frequent policy changes should not be unexpected as it is fundamentally a new feature provided by the system. Obviously, even these uncommon operations should be completed as fast as possible, but that has not been a major consideration in the current implementation.

6.2.4 Macrobenchmark A macrobenchmark evaluation of the Flask prototype is difficult to perform. Since Flask is a research prototype, it has only limited POSIX support and many of the servers are not robust or well tuned. As a result, it is difficult to run non-trivial benchmark applications. Nevertheless, we performed a simple comparison, running `make` to compile and link an application consisting of 20 `.c` and 4 `.h` files for a total of 8060 lines of code (including comments and white space), about 190KB total.

The test environment included three object managers (the kernel, BSD filesystem server and POSIX process manager) along with a shell and all the GNU utilities necessary to build the application (`make`, `gcc`, `ld`, etc.). The Flask configuration of the test includes the security server with the three object managers configured to include the security features described in Section 5.3 and Appendix A. For each configuration, we ran `make` five times, ignored the first run, and averaged the time of the final four runs (the initial run primed the data and metadata caches in the filesystem). To give a sense of the absolute performance of the base Fluke system, we also ran the test under FreeBSD 2.1.5 on the same machine and filesystem. Table 5 summarizes the experiment.

The slowdown for Flask over the base Fluke system is less than 5%. By running the Flask kernel with unmodified Fluke object managers (*Flask-FFS-PM*), we see that

OS Config	Time (sec)
BSD	18.6
Fluke	39.9
Flask	41.7 (4.5%)
Flask-FFS-PM	40.9 (2.5%)
Fluke-memfs	24.7
Flask-memfs	27.4 (11%)

Table 5: Results of running `make` to compile and link a simple application in various OS configurations. *BSD* is FreeBSD 2.1.5, *Flask-FFS-PM* is the Flask kernel with the unmodified Fluke filesystem server and process manager, and the *memfs* entries use a memory-based filesystem in place of the disk-based filesystem. Percentages are the slowdowns vs. the appropriate base Fluke configurations.

the overhead is pretty evenly divided between the kernel and the other object managers (primarily the filesystem server). However, this modest slowdown is against a Fluke system which is over twice as slow on the same test as a competitive Unix system (*BSD*). The bulk of this slowdown is due to the prototype filesystem server which does not do asynchronous or clustered I/O operations. To factor this out, we reran the tests using a memory-based filesystem which supports the same access checks as the disk-based filesystem. The last two lines of Table 5 show the results of these tests. Note that the Flask overhead has increased to 11%, as less is masked by the disk I/O latency.

Table 6 reports the number of security decisions that were requested by each object manager during testing of the *Flask* configuration and how those decisions were resolved. The numbers include all five runs of `make` as well as the intervening removal of the object files. These results reaffirm the effectiveness of caching security decisions, with well over 99% of the requests never reaching the security server.

6.2.5 Performance Conclusions Initial microbenchmark numbers suggest that the overhead of the Flask microkernel mechanisms can be made negligible through the use of the access vector cache and local hints when appropriate. They also highlight the need for an access vector cache so that communications with the security server and security computations within the security server are minimized. They also point to several areas for potential optimization, such as the AVC implementation, the communications infrastructure and the prototype security server computations. A complete analysis of the effectiveness of the AVC remains as future work. Issues such as the optimal cache size and the sensitivity of the AVC hit ratios to policy changes remain to be explored.

Results of the simple macrobenchmark test are inconclusive. Although the performance impact numbers are encouraging (5–11% slowdown), the bad absolute performance of the prototype system cannot be ignored.

Object	Total	Resolution		
Manager	queries	using hint	using cache	calling SS
Kernel	603735	175585	428121	29
FFS	76708	N/A	76700	8
PM	892	N/A	890	2

Table 6: Resolution of requested security decisions during the compilation benchmark. Numbers are from the *Flask* configuration of Table 5 and includes all five runs of `make` and `make clean`.

More completely exploring the performance overhead of the Flask security architecture remains as future work, and will likely be done in the context of a Linux or OS-Kit implementation of the architecture. This will permit more realistic workloads to be measured.

6.3 Scale and Invasiveness of Flask Code

In Table 7 we present data that give a rough estimate of the scale and complexity of adding fine-grained security enforcement to the base Fluke components. Overall, the Fluke components increased in size less than 8%. Although the kernel increased the most at 19%, for large object managers the percentage is reassuringly small (4–6%). Of these modifications, we examined the magnitude of changes involved by classifying each changed location as "trivial" changes (*e.g.*, one-line changes, `#define` changes, name or parameter changes, etc.) or "non-trivial." For the process manager, 57% of the changes fell into the trivial category. For the kernel, a similar percentage of the changes were trivial, 61%, despite the fact that the kernel is an order of magnitude larger and more complicated than the process manager.

The changes required to implement the Flask security architecture did not involve any modifications to the existing Fluke API. Extended calls were added to the existing API to permit security-aware applications to use the additional security functionality, such as the client and server identification support. All applications that run on the base Fluke system can be executed unchanged on Flask.

7 Summary

This paper describes an operating system security architecture capable of supporting a wide range of security policies, and the implementation of this architecture as part of the Flask microkernel-based operating system. It provides a usable definition of policy flexibility, identifies limitations of this definition and highlights the need for atomicity. It shows that capability systems and interposition techniques are inadequate for achieving policy flexibility. It presents the Flask architecture and describes how Flask overcomes the obstacles to achieving

Component	Fluke LOC	+Flask	%Incr.	#Locs.	%Locs.
Kernel	9271	1795	19.3	258	2.4
FFS	21802	1342	6.2	14	.06
Proc. Mgr	925	196	21.2	85	9.2
Net Server	24549	1071	4.4	224	9.1
Total	58435	4575	7.8	647	1.1

Table 7: "Filtered" source code size for various Flask components and the number of discrete locations in the base Fluke code that were modified. This count of source code lines filters out comments, blank lines, preprocessor directives, and punctuation-only lines, and typically is 1/4 to 1/2 the size of unfiltered code. The network server count includes the ISAKMP and IPSEC distributions, counting as modifications all Flask-specific changes to them and the base Fluke network component.

policy flexibility, including the need for atomicity. Although the performance evaluation of the Flask prototype is incomplete, this paper demonstrates that the architecture is practical to implement and flexible to use. Moreover, the architecture should be applicable to many other operating systems.

Availability

The Flask software and documentation are available at `<http://www.cs.utah.edu/flux/flask/>`.

A Other Flask object managers

This appendix describes the specific features that have been added to some of the Flask user-space object managers. Although the following subsections are not necessary for understanding the Flask architecture, they provide helpful insight into the details of providing policy flexibility in a complete system.

A.1 File Server

The Flask file server provides four types of controlled (labeled) objects: file systems, directories, files, and file description objects. Since file systems, directories and files are persistent objects, their labels must also be persistent. The binding of persistent labels to these objects is shown in Figure 6. The file server supports persistent labels without sacrificing policy flexibility or performance by treating security contexts as opaque strings and by mapping these labels to SIDs by a query to the security server for internal use in the file server. Control over file description objects is separated from control over the files themselves so that propagation of access to file description objects may be controlled by the policy. As noted in Section 3.1, the ability to control the propagation of access rights is critical to policy flexibility.

In contrast to the Unix file access controls, the Flask file server defines a permission for each service that observes or modifies the state of a file or directory. For

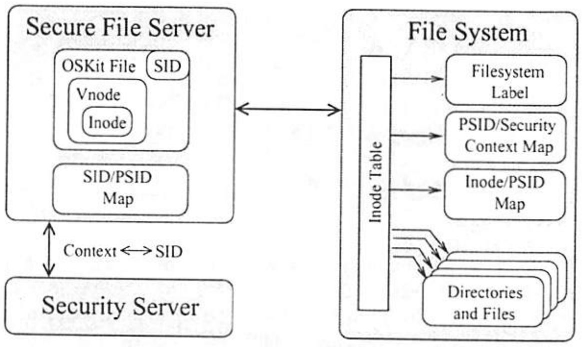


Figure 6: Labeling of persistent objects. The file server maintains a table within each file system which identifies the security context of the file system and every directory and file within the file system, thereby ensuring that the security attributes of these objects are preserved even if the file system is moved to another system. This table is partitioned into a mapping between each security context and an integer *persistent SID* (PSID) and a mapping between each object and its persistent SID. These persistent SIDs are purely an internal abstraction within the file system and have a distinct name space for each file system. Hence, persistent SIDs may be lightweight and the allocation of persistent SIDs may be optimized for each file system.

example, whereas Unix permits a process to invoke *stat* or *unlink* on a file purely on the basis of the process' access to the file's parent directory, the Flask file server checks *Getattr* and *Unlink* permissions to control access to the file itself in addition to the directory-based permissions. Such controls are necessary to generally support nondiscretionary security policies. The Flask file server also supports fine-grained distinctions among services, such as separate *Write* and *Append* permissions for files and separate *Add_name* and *Remove_name* permissions for directories, which is important for supporting policy flexibility.

The file server provides operations to relabel files and directories, since the relabel operation has the potential of being much more efficient than merely copying such objects into new objects with different labels. There are a couple of complications of relabeling. First, migrated permissions pertaining to the file may need to be revoked. For instance, changing the SID of a file may affect the permission to write to a file that is stored in a file description object. Hence, all such permissions are recomputed and revoked if necessary. Second, a relabeling operation cannot be simply controlled through the SID of the client subject and the SID of the file, but must also involve the newly requested SID. This is addressed by requiring three permissions for a relabel to complete, as shown in Table 8. The provision of a single relabel operation is also helpful from a policy flexibility perspective, since the policy logic can be directly expressed in terms of any of these three possible SID pairs. In contrast, implementing the same policy logic in terms of the permis-

SOURCE	TARGET	PERMISSION
Subject SID	File SID	RelabelFrom
Subject SID	New SID	RelabelTo
File SID	New SID	Transition

Table 8: Permission requirements for relabeling a file. Additionally, the subject must possess *Search* permission to every directory in the path.

sions controlling operations involved in copying an object would be complicated by the much weaker coupling among the relevant SIDs.

The file server design proposes the use of the Flask architecture's polyinstantiation support for *security union directories* (SUDs); however, the design for SUDs has not yet been implemented. SUDs are a generalization of the partitioned directory approach taken by multi-level secure Unix systems for dealing with /tmp. The SUD mechanism is designed to use the polyinstantiation support to determine the preferred member directory for each client to access by default. However, unlike the simple partitioned directory approach, the SUD mechanism provides a unified view of all accessible members within the polyinstantiated directory to clients based upon access decisions between the client and the member directories.

As was noted in Section 3.2, file server operations provide a simple example of the problems with implementing security controls at the server's external interface. The Flask file server draws its file system implementation from the OSKit [12] whose exported COM interfaces are similar to the internal VFS interface [27] used by many Unix file systems. It was possible to implement the Flask security controls at that interface where these problems do not exist.

A.2 Network Server

Abstractly, the Flask network server ensures that every network IPC is authorized by the security policy. Of course, a network server cannot independently ensure that a network IPC is authorized by the policy of its node, since it does not have end-to-end control over data delivery to processes on peer nodes. Instead, a network server must extend some level of trust to its peer network servers to enforce its own security policy, in combination with their own security policies, over the peer processes. This requires a reconciliation of security policies, which would be handled by a separate negotiation server. The current negotiation server is limited to negotiating network security protocols and cryptographic mechanisms using the ISAKMP [33] protocol. The precise form of trust and the precise level of trust extended to peer network servers can vary widely and would be defined within the policy. Extending the concept of policy flexibility to a networked environment will require such support for complex trust relationships.

SOURCE	TARGET	LAYER
Process SID	Socket SID	Socket
Message SID	Socket SID	Transport
Message SID Node SID	Node SID Net Interface SID	Network

Table 9: Layered controls in the network protocol stack. Each layer applies controls based upon the SIDs of the abstractions directly accessible at that layer. Node SIDs are provided to the network server by a separate network security server, which may query distributed databases for security attributes, and network interface SIDs may be locally configured.

The principal controlled object type for the network server is the socket. For socket types that maintain message boundaries (*e.g.*, datagram), the network server also binds a separate SID to each message sent or received on a socket. For other socket types, each message is implicitly associated with the SID of its sending socket. Since messages cross the boundary of control of the network server, and may even cross a policy domain boundary, the network server may need to apply cryptographic protections to messages in order to preserve the security requirements of the policy and must bind the security attributes of the message to the message. Our prototype network server uses the IPSEC [26] protocols for this purpose, with security associations established by the negotiation server. The negotiation server may not pass SIDs across the network, since they are only local identifiers; instead, the negotiation server must pass the actual security attributes to its peer, which can then establish its own SID for the corresponding security context. Although the negotiation server must handle security contexts, it does not interpret them, and thus remains policy-flexible. Attribute translation and interpretation must be performed by the corresponding security servers in accordance with the policy reconciliation.

The network server controls are layered to match the network protocol layering architecture. Hence, the abstract control over the high-level network IPC services consists of a collection of controls over the abstractions at each layer, as shown in Table 9. The layered controls provide the policy with the ability to precisely regulate network operations, using all the information relevant to security decisions, and they allow the policy to take advantage of specific characteristics of the different protocols (*e.g.*, the client/server relationship in TCP). The network server provides another example of the problems with implementing security controls at the server's external interface. This is due to the need to control abstractions and interpose on operations which are not exported by the network server's external interface.

Since the TCP and UDP port spaces are fixed resources, the network server uses the Flask architecture's polyinstantiation support for *security union port spaces* (SUPs). SUPs are analogous to the SUDs discussed in Section A.1. The polyinstantiation support is used to determine the preferred member port space when a port number is associated with a socket and when an incoming packet has a destination port number which exists in multiple member port spaces. The SUP mechanism provides a unified view of all accessible port spaces within the polyinstantiated port space based on access decisions.

Many of the details of the Flask network server and other servers that support it are beyond the scope of this paper. A much more detailed description of an earlier version of the Flask network server can be found in [9].

A.3 Process Manager

The Flask process manager implements the POSIX process abstraction, providing support for functions such as *fork* and *execve*. These higher-level process abstractions are layered on top of Flask processes, which consist of an address space and its associated threads. The process manager provides one controlled object type, the POSIX process, and binds a SID to each POSIX process. Unlike the SID of a Flask process, the SID of a POSIX process may change through an *execve*. Such SID transitions are controlled by the process *Transition* permission between the old and new SIDs. This control permits the policy to regulate a process' ability to transition to different security domains. Default transitions may be defined by the policy through the default object labeling mechanism described in Section 5.2.1.

In combination with the file server and the microkernel, the process manager is responsible for ensuring that each POSIX process is securely initialized. The file server ensures that the memory for the executable is labeled with the SID of the file. The microkernel ensures that the process may only execute memory to which it has *Execute* access. The process manager initializes the state of transformed POSIX processes, sanitizing their environment if the policy requires it.

Acknowledgments

We especially thank Jeff Turner for his many contributions to the Flask vision and architecture. Duane Olawsky contributed much to our understanding of the features required for policy flexibility. We also thank Dan Wallach, Grant Wagner, Andy Muckelbauer, Ruth Taylor, Charlie Payne, Tom Keefe and the anonymous reviewers for reviewing earlier drafts of this paper, Roland McGrath for recent Fluke implementation, Ajay Chitturi for implementing an earlier version of our secure network server design, and other members of the Flux group for help in numerous ways.

References

[1] M. D. Abrams. Renewed understanding of access control policies. In *Proceedings of the 16th National Computer Security Conference*, pages 87–96, Oct. 1993.

[2] M. D. Abrams, L. J. LaPadula, K. W. Eggers, and I. M. Olson. A generalized framework for access control: An informal description. In *Proceedings of the 13th National Computer Security Conference*, pages 135–143, Oct. 1990.

[3] D. E. Bell and L. J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA, May 1973.

[4] T. C. V. Benzel, E. J. Sebes, and H. Tajalli. Identification of subjects and objects in a trusted extensible client server architecture. In *Proceedings of the 18th National Information Systems Security Conference*, pages 83–99, 1995.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[6] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth National Computer Security Conference*, 1985.

[7] M. I. Bushnell. Towards a new strategy of OS design. *GNU's Bulletin*, 1(16), Jan. 1994.

[8] M. Carney and B. Loe. A comparison of methods for implementing adaptive security policies. In *Proceedings of the Seventh USENIX Security Symposium*, pages 1–14, Jan. 1998.

[9] A. Chitturi. Implementing mandatory network security in a policy-flexible system. Master's thesis, University of Utah, 1998. pp. 70. http://www.cs.utah.edu/flux/flask/.

[10] D. F. Ferraiolo, J. A. Cugini, and D. R. Kuhn. Role-Based Access Control (RBAC): Features and motivations. In *Proceedings of the Eleventh Annual Computer Security Applications Conference*, Dec. 1995.

[11] T. Fine and S. E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, May 1993.

[12] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

[13] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and Execution Models in the Fluke Kernel. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 101–116, Feb. 1999.

[14] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Symposium on Operating Systems Design and Implementations*, pages 137–151, Oct. 1996.

[15] T. Fraser and L. Badger. Ensuring continuity during dynamic security policy reconfiguration in dte. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 15–26, May 1998.

[16] M. Gasser. *Building a Secure Computer Systems*. Van Nostrand Reinhold Company, 1988.

[17] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, July 1996.

[18] L. Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, May 1989.

[19] R. Graubart. On the need for a third form of access control. In *Proceedings of the 12th National Computer Security Conference*, pages 296–304, Oct. 1989.

[20] R. Grimm and B. N. Bershad. Providing policy-neutral and transparent access control in extensible systems. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1999.

[21] N. Hardy. The confused deputy. *Operating Systems Review*, 22(4):36–38, Oct. 1988.

[22] T. Jaeger, J. Liedtke, and N. Islam. Operating system protection for fine-grained programs. In *Proceedings of the Seventh USENIX Security Symposium*, pages 143–157, Jan. 1998.

[23] R. Kain and C. Landwehr. On access checking in capability-based systems. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 66–77, May 1986.

[24] P. A. Karger. New methods for immediate revocation. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 48–55, May 1989.

[25] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, May 1984.

[26] S. Kent and R. Atkinson. Security architecture for the Internet Protocol. RFC 2401, Internet Engineering Task Force, Nov. 1998. ftp://ftp.isi.edu/in-notes/rfc2401.txt.

[27] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proc. of the Summer 1986 USENIX Conf.*, pages 238–247, Atlanta, GA, June 1986.

[28] C. R. Landau. Security in a secure capability-based system. *Operating Systems Review*, pages 2–4, Oct. 1989.

[29] R. Levin, E. Cohen, W. Corwin, P. F., and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 132–140, Unversity of Texas at Austin, Nov. 1975. ACM/SIGOPS.

[30] J. Liedtke. Clans and chiefs. In *Architektur von Rechensystemen*. Springer-Verlag, Mar. 1992.

[31] K. Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, Nov. 1992.

[32] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, Oct. 1998. http://-csrc.nist.gov/nissc/1998/proceedings/paperF1.pdf.

[33] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet security association and key management protocol (ISAKMP). RFC 2408, Internet Engineering Task Force, Nov. 1998. ftp://-ftp.isi.edu/in-notes/rfc2408.txt.

[34] C. J. McCollum, J. R. Messing, and L. Notargiacomo. Beyond the pale of mac and dac - defining new forms of access control. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 190–200, May 1990.

[35] S. E. Minear. Providing policy control over object operations in a Mach based system. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, June 1995.

[36] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An overview of the Spring system. In *A Spring Collection*. Sun Microsystems, Inc., 1994.

[37] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proceedings of the Annual Computer Security Applications Conference*, Dec. 1997.

[38] D. Olawsky, T. Fine, E. Schneider, and R. Spencer. Developing and using a "policy neutral" access control policy. In *Proceedings of the New Security Paradigms Workshop*. ACM, Sept. 1996.

[39] E. I. Organick. *The Multics System : An Examination of its Structure*. MIT Press, 1972.

[40] S. A. Rajunas, N. Hardy, A. C. Bomberger, W. S. Frantz, and C. R. Landau. Security in KeyKOS. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 78–85, Apr. 1986.

[41] S. G. Ravi Sandhu, Venkata Bhamidipati and C. Youman. The arbac97 model for role-based administration of roles: Preliminary description and outline. In *Proceedings of the Second ACM Workshop on Role-Based Access Control*, pages 41–50, Nov. 1997.

[42] D. Redell and R. Fabry. Selective revocation of capabilities. In *Proceedings of the International Workshop on Protection in Operating Systems*, pages 192–209, Aug. 1974.

[43] Secure Computing Corp. DTOS Generalized Security Policy Specification. DTOS CDRL A019, 2675 Long Lake Rd, Roseville, MN 55113, June 1997. http://-www.securecomputing.com/randt/HTML/dtos.html.

[44] Secure Computing Corp. Assurance in the Fluke Microkernel: Formal Security Policy Model. CDRL A003, 2675 Long Lake Rd, Roseville, MN 55113, Feb. 1999. http://www.cs.utah.edu/-flux/flask/.

[45] Secure Computing Corp. Assurance in the Fluke Microkernel: Formal Top-Level Specification. CDRL A004, 2675 Long Lake Rd, Roseville, MN 55113, Feb. 1999. http://www.cs.utah.edu/-flux/flask/.

[46] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996. USENIX Assoc.

[47] J. S. Shapiro. EROS: A capability system. Technical Report Technical Report MS-CIS-97-04, University of Pennsylvania, Department of Computer and Information Science, 1997.

[48] D. F. Sterne, M. Branstad, B. Hubbard, and B. M. D. Wolcott. An analysis of application specific security policies. In *Proceedings of the 14th National Computer Security Conference*, pages 25–36, Oct. 1991.

[49] SunSoft, Inc. *Spring Programmer's Guide*, 1995. On-line documentation included in the Spring Research Distribution 1.0.

[50] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 116–128, Oct. 1997.

[51] R. M. Wong. A comparison of secure unix operating systems. In *Proceedings of the Sixth Annual Computer Security Applications Conference*, pages 322–333, Dec. 1990.

[52] W. Wulf, R. Levin, and P. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[53] M. E. Zurko and R. Simon. User-centered security. In *Proceedings of the New Security Paradigms Workshop*, Sept. 1996.

A Study in Using Neural Networks for Anomaly and Misuse Detection*

Anup K. Ghosh & Aaron Schwartzbard
Reliable Software Technologies
21351 Ridgetop Circle, Suite 400
Dulles, VA 20166
poc:aghosh@rstcorp.com
http://www.rstcorp.com

Abstract

Current intrusion detection systems lack the ability to generalize from previously observed attacks to detect even slight variations of known attacks. This paper describes new process-based intrusion detection approaches that provide the ability to generalize from previously observed behavior to recognize future unseen behavior. The approach employs artificial neural networks (ANNs), and can be used for both anomaly detection in order to detect novel attacks and misuse detection in order to detect known attacks and even variations of known attacks. These techniques were applied to a large corpus of data collected by Lincoln Labs at MIT for an intrusion detection system evaluation sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA). Results from applying these techniques for both anomaly and misuse detection against the DARPA evaluation data are presented.

1 Introduction

Results from a recent U.S. Defense Advanced Research Projects Agency (DARPA) study highlight the strengths and weaknesses of current research approaches to intrusion detection. The DARPA scientific study is the first of its kind to provide independent third party evaluation of intrusion detection tools against such a large corpus of data. The findings from this study indicate that a fundamental paradigm shift in intrusion detection research is necessary to provide reasonable levels of detection against novel attacks and even variations of known attacks. Central to this goal is the ability to generalize from previously observed behavior to recognize future unseen, but similar behavior. To this end, this paper describes a study in using neural networks for both anomaly detection and misuse detection.

Research in intrusion detection research has begun to shift from analyzing user behavior to analyzing process behavior. Initial work in analyzing process behavior has already shown promising results in providing very high levels of detection against certain classes of attacks. In particular, process-based anomaly detection approaches have shown very good performance against novel attacks that result in unauthorized local access and attacks that result in elevated privileges — a vulnerable area for most intrusion detection tools [6]. In spite of the good detection capability of process-based anomaly detection approaches, the results indicate high rates of false alarms that can make these tools unusable for the practical security administrator. Current wisdom is that false alarm rates must be reduced to the level of one to two false alarms per day in order to make the system usable by administrators.

One of the largest challenges for today's intrusion detection tools is being able to generalize from previously observed behavior (normal or malicious) to recognize similar future behavior. This problem is acute for signature-based misuse detection approaches, but also plagues anomaly detection tools that must be able to recognize future normal behavior that is not identical to past observed behavior,

*This work was funded by the Defense Advanced Research Projects Agency (DARPA) under Contract DAAH01-97-C-R095. THE VIEWS AND CONCLUSIONS CONTAINED IN THIS DOCUMENT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY OR THE U.S. GOVERNMENT.

in order to reduce false positive rates.

To address this shortcoming, we utilize a simple neural network that can generalize from past observed behavior to recognize similar future behavior. In the past, we have applied backpropagation networks in addition to other neural networks with good performance to the problem of anomaly detection [8]. Here we present using a neural network for both anomaly and misuse detection. The approach is evaluated against the DARPA intrusion detection evaluation data.

2 Prior Art in Intrusion Detection

Some of the earliest work in intrusion detection was performed by Jim Anderson in the early 1980s [1]. Anderson defines an intrusion as any unauthorized attempt to access, manipulate, modify, or destroy information, or to render a system unreliable or unusable. Intrusion detection attempts to detect these types of activities. In this section we establish the foundations of intrusion detection techniques in order to determine where they are strong and where they need improvement.

2.1 Anomaly detection vs. misuse detection

Intrusion detection techniques are generally classified into two categories: anomaly detection and misuse detection. Anomaly detection assumes that misuse or intrusions are highly correlated to abnormal behavior exhibited by either a user or the system. Anomaly detection approaches must first baseline the normal behavior of the object being monitored, then use deviations from this baseline to detect possible intrusions. The initial impetus for anomaly detection was suggested by Anderson in his 1980 technical report when he noted that intruders can be detected by observing departures from established patterns of use for individual users. Anomaly detection approaches have been implemented in expert systems that use rules for normal behavior to identify possible intrusions [15], in establishing statistical models for user or program profiles [6, 4, 22, 19, 16, 18, 17], and in using machine learning to recognize anomalous user or program behavior [10, 5, 2, 14].

Misuse detection techniques attempt to model attacks on a system as specific patterns, then systematically scan the system for occurrences of these patterns. This process involves a specific encoding of previous behaviors and actions that were deemed intrusive or malicious. The earliest misuse detection methods involved off-line analysis of audit trails normally recorded by host machines. For instance, a security officer would manually inspect audit trail log entries to determine if failed root login attempts were recorded. Manual inspection was quickly replaced by automated analysis tools that would scan these logs based on specific patterns of intrusion. Misuse detection approaches include expert systems [15, 3], model-based reasoning [13, 7], state transition analysis [23, 12, 11, 21], and keystroke dynamics monitoring [20, 13]. Today, the vast majority of commercial and research intrusion detection tools are misuse detection tools that identify attacks based on attack signatures.

It is important to establish the key differences between anomaly detection and misuse detection approaches. The most significant advantage of misuse detection approaches is that known attacks can be detected fairly reliably and with a low false positive rate. Since specific attack sequences are encoded into misuse detection systems, it is very easy to determine exactly which attacks, or possible attacks, the system is currently experiencing. If the log data does not contain the attack signature, no alarm is raised. As a result, the false positive rate can be reduced very close to zero. However, the key drawback of misuse detection approaches is that they cannot detect novel attacks against systems that leave different signatures. So, while the false positive rate can be made extremely low, the rate of missed attacks (false negatives) can be extremely high depending on the ingenuity of the attackers. As a result, misuse detection approaches provide little defense against novel attacks, until they can learn to generalize from known signatures of attacks.

Anomaly detection techniques, on the other hand, directly address the problem of detecting novel attacks against systems. This is possible because anomaly detection techniques do not scan for specific patterns, but instead compare current activities against statistical models of past behavior. Any activity sufficiently deviant from the model will be flagged as anomalous, and hence considered as a possible attack. Furthermore, anomaly detection schemes are based on actual user histories and system data to create its internal models rather than pre-defined patterns. Though anomaly detection approaches are powerful in that they can detect novel attacks, they have their drawbacks as well. For instance, one clear drawback of anomaly detection is its inability to identify the specific type of attack that is occurring. However, probably the most significant disadvantage of anomaly detection approaches is the high rates of false alarm. Because any significant deviation from the baseline can be flagged as an intrusion, non-intrusive behavior that falls outside the normal range will also be labeled as an intrusion — resulting in a false positive. Another drawback of anomaly detection approaches is that if an attack occurs during the training period for establishing the baseline data, then this intrusive behavior will be established as part of the normal baseline. In spite of the potential drawbacks of anomaly detection, having the ability to detect novel attacks makes anomaly detection a requisite if future, unknown, and novel attacks against computer systems are to be detected.

2.2 Assessing the Performance of Current IDSs

In 1998, the U.S. Defense Advanced Research Projects Agency (DARPA) initiated an evaluation of its intrusion detection research projects.[1] To date, it is the most comprehensive scientific study known for comparing the performance of different intrusion detection systems (IDSs). MIT's Lincoln Laboratory set up a private controlled network environment for generating and distributing sniffed network data and audit data recorded on host machines. Network traffic was synthesized to replicate normal traffic as well as attacks seen on example military installations. Because all the data was generated, the laboratory has *a priori* knowledge of which data is normal and which is attack data. The simulated network represented thousands of internal Unix hosts and hundreds of users. Network traffic was generated to represent the following types of services: http, smtp, POP3, FTP, IRC, telnet, X, SQL/telnet, DNS, finger, SNMP, and time. This corpus of data is the most comprehensive set known to be generated for the purpose of evaluating intrusion detection systems and represents a significant advancement in the scientific community for independently and scientifically evaluating the performance of any given intrusion detection system.

[1]See `www.ll.mit.edu/IST/ideval/index.html` for a summary of the program.

TCP/IP data was collected using a network sniffer and host machine audit data was collected using Sun Microsystem's Solaris Basic Security Module (BSM). In addition, dumps of the file system from one of the Solaris hosts were provided. This data was distributed to participating project sites in two phases: training data and test data. The training data is data labeled as normal or attack and is used by the participating sites to train their respective intrusion detection systems. Once trained, the test data is distributed to participating sites in unlabeled form. That is, the participating sites do not know *a priori* which data in the test data is normal or attack. The data is analyzed off-line by the participating sites to determine which sessions are normal and which constitute intrusions. The results were sent back to MIT's Lincoln Labs for evaluation.

The attacks were divided into four categories: denial of service, probing/surveillance, remote to local, and user to root attacks. Denial of service attacks attempt to render a system or service unusable to legitimate users. Probing/surveillance attacks attempt to map out system vulnerabilities and usually serve as a launching point for future attacks. Remote to local attacks attempt to gain local account privilege from a remote and unauthorized account or system. User to root attacks attempt to elevate the privilege of a local user to root (or super user) privilege. There were a total of 114 attacks in 2 weeks of test data including 11 types of DoS attacks, 6 types of probing/surveillance attacks, 14 types of remote to local attacks, 7 types of user to root attacks, and multiple instances of all types of attacks.

The attacks in the test data were also categorized as old versus new and clear versus stealthy. An attack is labeled as old if it appeared in the training data and new if it did not. When an attempt was made to veil an attack, it was labeled as stealthy, otherwise it was labeled as clear.

The reason we present this evaluation study is because we believe it to represent the true state of the art in intrusion detection research. As such, it represents the foundation of more than 10 years of intrusion detection research upon which all future work in intrusion detection should improve. From this study, we can learn the strengths of current intrusion detection approaches, and more importantly, their weaknesses. Rather than identifying which systems performed well and which did not, we simply summarize the results of the overall best combination system.

Lincoln Laboratory reported that if the best performing systems against all four categories of attacks were combined into a single system, then roughly between 60 to 70 percent of the attacks would have been detected with a false positive rate of lower than 0.01%, or lower than 10 false alarms a day. This result summarizes the combination of best systems against all of the attacks simulated in the data. It shows that even in the best case scenario over 30% of the simulated attacks would go by undetected. However, the good news is that the false alarm rate is acceptably low — low enough that the techniques can scale well to large sites with lots of traffic. The bad news is that with over 30% of attacks going undetected in the best combination of current intrusion detection systems, the state of the art in intrusion detection does not adequately address the threat of computer-based attacks.

Further analysis showed that most of the systems reliably detected old attacks that occurred within the training data with low false alarm rates. These results apply primarily to the network-based intrusion detection systems that processed the TCP/IP data. This result is encouraging, but not too surprising since most of the evaluated systems were network-based misuse detection systems. The results were mixed in detecting new attacks. In two categories of attacks, probing/surveillance and user to root attacks, the performance in detecting new attacks was comparable to detecting old attacks. In the other two categories — denial of service and remote to local attacks — the performance of the top three network-based intrusion systems was roughly 20% detection for new denial of service attacks and less than 10% detection for new remote to local attacks. Thus, the results show that the best of today's network-based intrusion detection systems do not detect novel denial of service attacks nor novel remote to local attacks — arguably two of the most concerning types of attacks against computer systems today.

3 Monitoring Process Behavior for Intrusion Detection

In the preceding section, intrusion detection methods were categorized into either misuse detection or anomaly detection approaches. In addition, intrusion detection tools can be further divided into network-based or host-based intrusion detection. The distinction is useful because network-based intrusion detection tools usually process completely different data sets and features than host-based intrusion detection. As a result, the types of attacks that are detected with network-based intrusion detection tools are usually different than host-based intrusion detection tools. Some attacks can be detected by both network-based and host-based IDSs, however, the "sweet spots", or the types of attacks each is best at detecting, are usually distinct. As a result, it is difficult to make direct comparisons between the performance of a network-based IDS and a host-based IDS. A useful corollary of distinct sweetspots, though, is that in combination both techniques are more powerful than either one by itself.

Recent research in intrusion detection techniques has shifted the focus from user-based intrusion detection to process-based intrusion detection. Process-based monitoring intrusion detection tools analyze the behavior of executing processes for possible intrusive activity. The premise of process monitoring for intrusion detection is that most computer security violations are made possible by misusing programs. When a program is misused its behavior will differ from its normal usage. Therefore, if the behavior of a program can be adequately captured in a compact representation, then the behavioral features can be used for intrusion detection.

Two possible approaches to monitoring process behavior are: instrumenting programs to capture their internal states or monitoring the operating system to capture external system calls made by a program. The latter option is more attractive in general because it does not require access to source code for instrumentation. As a result, analyzing external system calls can be applied to commercial off the shelf (COTS) software directly. Most modern day operating systems provide built-in instrumentation hooks for capturing a particular process's system calls. On Linux and other variants of Unix, the `strace(1)` program allows one to observe system calls made by a monitored process as well as their return values. On Sun Microsystem's Solaris operating system, the Basic Security Module (BSM) produces an event record for individual processes. BSM recognizes 243 built-in system signals that can be made by a process. Thus, on Unix systems, there is good built-in support for tracing processes' externally observable behavior. Windows NT currently lacks a built-in auditing facility that provides such fine-grain resolution of program behavior.

Most process-based intrusion detection tools are based on anomaly detection. A normal profile for program behavior is built during the training phase of the IDS by capturing the program's system calls during normal usage. During the detection phase, the profile of system calls captured during on-line usage is compared against the normal profile. If a significant deviation from the normal profile is noted, then an intrusion flag is raised.

Early work in process monitoring was pioneered by Stephanie Forrest's research group out of the University of New Mexico. This group uses the analogy of the human immune system to develop intrusion detection models for computer programs. As in the human immune system, the problem of anomaly detection can be characterized as the problem of distinguishing between self and dangerous non-self [6]. Thus, the intrusion detection system needs to build an adequate profile of self behavior in order to detect dangerous behavior such as attacks. Using `strace(1)` on Linux, the UNM group analyzed short sequences of system calls made by programs to the operating system [6].

More recently, a similar approach was employed by the authors in analyzing BSM data provided under the DARPA 1998 Intrusion Detection Evaluation program [9]. The study compiled normal behavior profiles for approximately 150 programs. The profile for each program is stored in a table that consists of short sequences of system calls. During on-line testing, short sequences of system calls captured by the BSM auditing facility are looked up in the table. This approach is known as equality matching. That is, if an exact match of the sequence of system calls captured during on-line testing exists in the program's table, then the behavior is considered normal. Otherwise an anomaly counter is incremented.

The data is partitioned into fixed-size windows in order to exploit a property of attacks that tends to leave its signature in temporally co-located events. That is, attacks tend to cause anomalous behavior to be recorded in groups. Thus, rather than averaging the number of anomalous events recorded over the entire execution trace (which might wash out an attack in the noise), a much smaller size window of events is used for counting anomalous events.

Several counters are kept at varying levels of granularity from a counter for each fixed window of system calls to a counter for the number of windows that are anomalous. Thresholds are applied at each level to determine at which point anomalous behavior is propagated up to the next level. Ultimately, if enough windows of system calls in a program are deemed anomalous, the program behavior during a particular session is deemed anomalous, and an intrusion detection flag is raised.

The results from the study showed a high rate of detection, if not a low false positive rate [9]. Despite the simplicity of the approach and the high levels of detection, there are two main drawbacks to the equality matching approach: (1) large tables of program behavior must be built for each program, and (2) the equality matching approach does not have the ability to recognize behavior that is similar, but not identical to past behavior. The first problem becomes an issue of storage requirements for program behavior profiles and is also a function of the number of programs that must be monitored. The second problem results from the inability of the algorithm to generalize from past observed behavior. The problem is that behavior that is normal, yet slightly different from past recorded behavior, will be recorded as anomalous. As a result, the false positive rate could be artificially elevated. Instead, it is desirable to be able to recognize behaviors that are similar to normal, but not necessarily identical to past normal behavior as normal. Likewise, the same can be said for a misuse detection system. Many misuse detection systems are trained to recognize attacks based on exact signatures. As a result, slight variations among a given attack can result in missed detections, leading to a lower detection rate. It is desirable for misuse detection systems to be able to generalize from past observed attacks to recognize future attacks that are similar.

To this end, the research described in the rest of the paper employs neural networks to generalize from previously observed behavior. We develop an anomaly detection system that uses neural networks to learn normal behavior for programs. The trained network is then used to detect possibly intrusive behavior by identifying significant anomalies. Similarly, we developed a misuse detection system to learn the behavior of programs under attack scenarios. This system is then used to detect future attacks against the system. The goal of these approaches is to be able to recognize known attacks and detect novel attacks in the future. By using the associative connections of the network, we can generalize from past observed behavior to recognize future similar behavior. A comparison of the two systems against the DARPA intrusion data is provided in Section 5.

4 Using Neural Networks for Intrusion Detection

Applying machine learning to intrusion detection has been developed elsewhere as well [5, 2, 14]. Lane and Brodley's work uses machine learning to distinguish between normal and anomalous behavior. However, their work is different from ours in that they build *user* profiles based on sequences of each individual's normal user commands and attempt to detect intruders based on deviations from the established user profile. Similarly, Endler's work [5] used neural networks to learn the behavior of users based on BSM events recorded from user actions. Rather than building profiles on a per-user basis, our work builds profiles of *software behavior* and attempts to distinguish between normal software behavior and malicious software behavior. The advantages of our approach are that vagaries of individual behavior are abstracted because program behavior rather than individual usage is studied. This can be of benefit for defeating a user who slowly changes his or her behavior to foil a user profiling system. It can also protect the privacy interests of users from a surveillance system that monitors a user's every move.

The goal in using artificial neural networks (ANNs) for intrusion detection is to be able to generalize from incomplete data and to be able to classify online data as being normal or intrusive. An artificial neural network is composed of simple processing units, or *nodes*, and connections between them. The connection between any two units has some *weight*, which is used to determine how much one unit will affect the other. A subset of the units of the network acts as *input nodes*, and another subset acts as *output nodes*. By assigning a value, or *activation*, to each input node, and allowing the activations to propagate through the network, a neural network performs a functional mapping from one set of values (assigned to the input nodes) to another set of values (retrieved from the output nodes). The mapping itself is stored in the weights of the network.

In this work, a classical feed-forward multi-layer perceptron network was implemented: a backpropagation neural network. The backpropagation network has been used successfully in other intrusion detection studies [10, 2]. The backpropagation network, or backprop, is a standard feedforward network. Input is submitted to the network and the activations for each level of neurons are cascaded forward.

Our previous research in intrusion detection with BSM data used an equality matching technique to look up currently observed program behavior that had been previously stored in a table. While the results were encouraging, we also realized that the equality matching approach had no possibility of generalizing from previously observed behavior. As a result, we are pursuing research in using artificial neural networks to accomplish the same goals, albeit with better performance. Specifically, we are interested in the capability of ANNs to generalize from past observed behavior to detect novel attacks against systems. To this end, we constructed two different ANNs: one for anomaly detection and one for misuse detection.

To use the backprop networks, we had to address five major issues: how to encode the data for input to the network, what network topology should be used, how to train the networks, how to perform anomaly detection with a supervised training algorithm, and what to do with the data produced by the neural network.

Encoding the data to be used with the neural network is in general, a difficult problem. Previous experiments indicated that strings of six consecutive BSM events carried enough implicit information to be accurately distinguished as anomalous or normal for programs in general. One possible encoding technique was simply to enumerate all observed strings of six BSM events, and use the enumeration as an encoding. However, part of the motivation of using neural nets was their ability to classify novel inputs based on similarity to known inputs. A simple enumeration will fail to capture information about the strings. Therefore, a neural net will be less likely to be able to correctly classify novel inputs. In order to capture the necessary information in the encoding, we devised a distance metric for strings of events. The distance metric took into account the events common to two strings, as well as the difference in positions of common events. To encode a string of data, the distance metric was used to measure the distance from the data string to each of several "exemplar" strings. The encoding then consisted of a set of measured distances. A string could then be thought of as a point in a space where each dimension corresponded to one of the exemplar strings, and the point is mapped in the space by plotting the distance from each dimension.

Once an appropriate encoding method was developed, an appropriate network topology must be employed. We had to determine how many input and output nodes were necessary, and if a hidden layer was to be used, how many nodes should it contain. Because we seek to determine whether an input string is anomalous or normal, we use a single continuously valued output node to represent the extent to which the network believes the input is normal or anomalous. The more anomalous the input is, the closer to 1.0 the network computes its output. Conversely, the closer to normal the input is, the closer to 0.0, the output node computes.

The number of input nodes has to be equal to the number of exemplar strings (since each exemplar produced a distance for input to the network). With an input layer, a hidden layer, and an output layer, a neural network can be constructed to compute any arbitrarily complex function. Thus, a single hidden layer was used in our networks. A different network must be constructed, tuned, and trained for each program to be monitored, since what might have been quite normal behavior for one program might have been extremely rare in another. The number of hidden nodes varied based on the performance of each trained network.

During training, many networks were trained for each program, and the network that performed the best was selected. The remaining networks were discarded. Training involved exposing the networks to four weeks of labeled data, and performing the backprop algorithm to adjust weights. An epoch of training consisted of one pass over the training data. For each network, the training proceeded until the total error made during an epoch stopped decreasing, or 1,000 epochs had been reached. Since the optimal number of hidden nodes for a program was not known before training, for each program, networks were trained with 10, 15, 20, 25, 30, 35, 40, 50, and 60 hidden nodes. Before training, network weights were initialized randomly. However, initial weights can have a large, but unpredictable, effect on the performance of a trained network. In order to avoid poor performance due to bad initial weights, for each program, for each number of hidden nodes, 10 networks were initialized differently, and trained. Therefore, for each program, 90 networks were trained. To select which of the 90 to keep, each was tested on two weeks of data that were not part of the four weeks of data used for training. The network that classified data most accurately was kept.

4.1 Anomaly detection

In order to train the networks, it is necessary to expose them to normal data and anomalous data. Randomly generated data was used to train the network to distinguish between normal and anomalous data. The randomly generated data, which were spread throughout the input space, caused the network to generalize that all data were anomalous by default. The normal data, which tended to be localized in the input space, caused the network to recognize a particular area of the input space as non-anomalous.

After training and selection, a set of neural networks was ready to be used. However, a neural network can only classify a single string (a sequence of BSM events) as anomalous or normal, and our intention was to classify entire sessions (which are usually composed of executions of multiple programs) as anomalous or normal. Furthermore, our previous experiments showed that it is important to capture the temporal locality of anomalous events in order to recognize intrusive behavior. As a result, we desired an algorithm that provides some memory of recent events.

The leaky bucket algorithm fits this purpose well. The leaky bucket algorithm keeps a memory of recent events by incrementing a counter of the neural network's output, while slowly leaking its value. Thus, as the network computes many anomalies, the leaky bucket algorithm will quickly accumulate a large value in its counter. Similarly, as the network computes a normal output, the bucket will "leak" away its anomaly counter back down to zero. As a result, the leaky bucket emphasizes anomalies that are closely temporally co-located and diminishes the values of those that are sparsely located.

Strings of BSM events are passed to a neural network in the order they occurred during program execution. The output of a neural network (that is, the classification of the input string) is then placed into a leaky bucket. During each timestep, the level of the bucket is decreased by a fixed amount. If the level in the bucket rises above some threshold at any point during execution of the program, the program is flagged as anomalous. The advantage of the using a leaky bucket algorithm is that it allows occasional anomalous behavior, which is to be expected during normal system operation, but it is quite sensitive to large numbers of temporally co-located anomalies, which one would expect if a program were really being misused. If a session contains a single anomalous execution of a program, the session is flagged as anomalous.

4.2 Misuse detection

Having developed a system for anomaly detection, we chose to evaluate how well the same techniques could be applied to misuse detection. Our system is designed to recognize some *type* of behavior. Thus, it should not matter whether the behavior it is learning was normal system usage, or attack behavior. Aside from trivial changes to the way the leaky bucket is monitored, our system should not require any modification to perform misuse detection. Having made the trivial modification to the leaky bucket, we tested our system as a misuse detector.

Unfortunately, two issues particular to our data-set made misuse detection difficult. The first issue was a lack of data. In the DARPA data, there was between two to three orders in magnitude less intrusion data than normal data. This made it quite difficult to train networks to learn what constituted an attack. The second issue was related to the labeling of intrusions. Intrusion data were labeled on a session-by-session basis. Whereas several programs might be executed during an intrusive session, as few as one might be anomalous. Thus, while all data labeled non-intrusive could be assumed to be normal, not all data labeled intrusive could be assumed to be anomalous. Despite these stumbling blocks, we configured our neural network system for misuse detection.

5 Experimental Results

The anomaly and misuse detection systems were tested on the same test data. The test data consisted of 139 non-intrusive sessions, and 22 intrusive sessions. Although it would have been preferable to use a larger number of intrusive sessions for testing, there were so few intrusive sessions in the DARPA data that all other intrusion data were used to train the misuse detection system.

The performance of any intrusion detection system must account for both the detection ability and the false positive rate. We observed both of these factors while varying the leak rate used by the leaky bucket algorithm. A leak rate of 0 results in all prior timesteps being retained in memory. A leak rate of 1 results in all timesteps but the current one being forgotten. We varied the leak rate from 0 to 1.

The performance of the IDS should by judged in terms of both the ability to detect intrusions, and by false positives—incorrect classification of secure behavior as insecure. We used receiver operating characteristic (ROC) curves to compare intrusion detection ability to false positives. A ROC curve is a parametric plot, where the parameter is the sensitivity of the system to what it perceives to be insecure behavior. The curve is a plot of the likelihood that an intrusion is detected, against the likelihood that a non-intrusion is misclassified for a particular parameter, such as a threshold. The ROC curve can be used to determine the performance of the system for any possible operating point. The ROC curve allows the end user of an intrusion detection system to assess the trade-off between detection ability and false alarm rate in order to properly tune the system for acceptable tolerances.

Different leak rates produced different ROC curves. Figure 1 displays two ROC curves—one for a low leak rate, and one for a high leak rate. For the leak rate of .2, to achieve detection better than 77.3%, one must be willing to accept a dramatic increase in false positives. At 77.3% detection, the false positive rate is only 3.6%. When the leak rate is .7, a detection rate of 77.3% can be achieved with a false positive rate of only 2.2%.

ROC curves were also produced for the performance of our misuse detection system. While the performance was not nearly as good as the anomaly detection system in terms of false positives (which was a high as 5% for even low sensitivity rates), the misuse detection system displayed very high detection abilities—especially surprising due to the small number of sessions used to train the system. As illustrated in Figure 2, with a leak rate of 0.7, the sys-

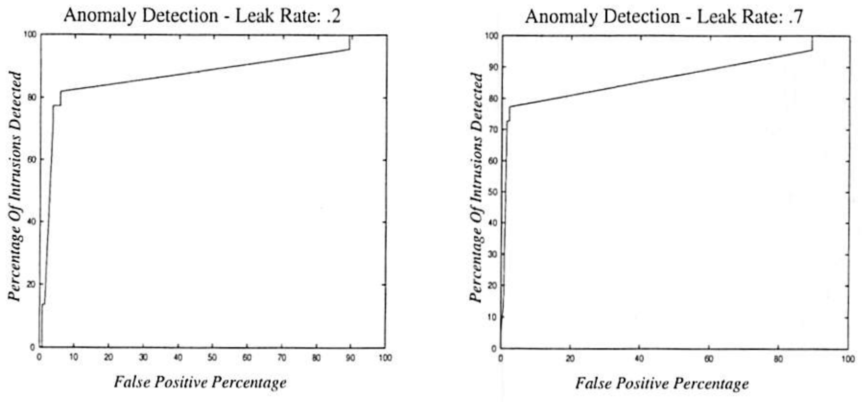


Figure 1: Anomaly detection results for two different leak rates.

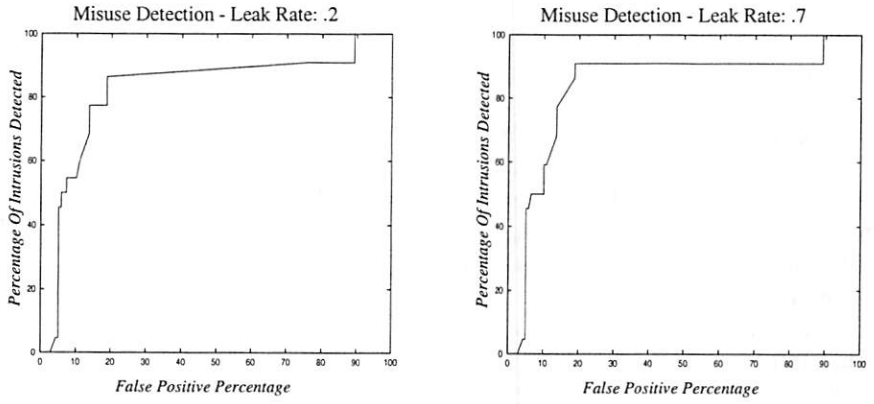


Figure 2: Misuse detection results for two different leak rates.

tem was able to detect as much as 90.9% of all intrusions with a false positive rate of 18.7%. Other host-based misuse detection systems can currently provide similar detection capabilities with lower false positive rates. Thus, this approach to misuse detection may not be suitable for detecting attacks in comparison to signature-based approaches. However, our technique demonstrated the ability of the system to detect novel attacks by generalizing from previously observed behavior.

While that false positive rate is clearly unacceptable, it should be remembered that the misuse detection system was trained on data which contained not only intrusion data, but also normal data. This would naturally lead this system to produce a large amount of false positives. By eliminating the non-intrusion data from the training data, it is believed that significantly lower false positive rates could be achieved, without lowering the detection ability.

The results of our experiments indicate that neural networks are suited to perform intrusion detection and can generalize from previously observed behavior. Currently, the false positive rates are too high to be practical for commercial users. In order to be a useful tool, false positive rates need to be between one and three orders of magnitude smaller. We continue to investigate how to improve the performance of our neural networks. Tests of variations on our techniques indicate that we have not yet achieved optimal performance.

6 Conclusions

This paper began with an examination of current intrusion detection systems. In particular, the DARPA 1998 intrusion detection evaluation study found that novel attacks against systems are rarely detected by most IDSs that use signatures for detection. On the other hand, well-known attacks for which signatures from network data or host-based intrusion data can be formed, perform very reliably with low rates of false alarms. However, even slight variations of known attacks escape detection by signature-based IDSs. Similarly, program-based anomaly detection systems have performed very well in detecting novel attacks, albeit with high false alarm rates. To overcome the problems in current misuse detection and anomaly detection approaches, a key necessity for IDSs is the ability to generalize from previously observed behavior to recognize future similar behavior. This capbility will permit detection of variations of known attacks as well as reduce false positive rates for anomaly-based IDSs.

In this paper, we presented the application of a simple neural network to learning previously observed behavior in order to detect future intrusions against systems. The results from our study show the viability of our approach for detecting intrusions. Future work will apply other neural networks more suited toward the problem domain of analyzing temporal characteristics of program traces. For instance, applying recurrent, time delay neural networks to program-based anomaly detection has proved to be more successful than using backpropagation networks for the same purpose [8]. Our next step is to apply these networks to misuse detection as well.

References

[1] J.P. Anderson. Computer security threat monitoring and surveillance. Technical Report Technical Report, James P. Anderson Co., Fort Washington, PA, April 1980.

[2] J. Cannady. Artificial neural networks for misuse detection. In *Proceedings of the 1998 National Information Systems Security Conference (NISSC'98)*, pages 443–456, October 5-8 1998. Arlington, VA.

[3] W.W. Cohen. Fast effective rule induction. In *Machine Learning: Proceedings of the Twelfth International Conference.* Morgan Kaufmann, 1995.

[4] P. D'haeseleer, S. Forrest, and P. Helman. An immunological approach to change detection: Algorithms, analysis and implications. In *IEEE Symposium on Security and Privacy*, 1996.

[5] D. Endler. Intrusion detection: Applying machine learning to solaris audit data. In *Proceedings of the 1998 Annual Computer Security Applications Conference (ACSAC'98)*, pages 268–279, Los Alamitos, CA, December 1998. IEEE Computer Society, IEEE Computer Society Press. Scottsdale, AZ.

[6] S. Forrest, S.A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

[7] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, October 1991.

[8] A.K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring.* USENIX Association, April 11-12 1999. To appear.

[9] A.K. Ghosh, A. Schwartzbard, and M. Schatz. Using program behavior profiles for intrusion detection. In *Proceedings of the SANS Intrusion Detection Workshop*, February 1999. To appear.

[10] A.K. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings of the 1998 Annual Computer Security Applications Conference (ACSAC'98)*, December 1998.

[11] K. Ilgun. Ustat: A real-time intrusion detection system for unix. Master's thesis, Computer Science Dept, UCSB, July 1992.

[12] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State transition analysis: A rule-based intrusion detection system. *IEEE Transactions on Software Engineering*, 21(3), March 1995.

[13] S. Kumar and E.H. Spafford. A pattern matching model for misuse intrusion detection. The COAST Project, Purdue University, 1996.

[14] T. Lane and C.E. Brodley. An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conference*, pages 366–377, October 1997.

[15] W. Lee, S. Stolfo, and P.K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of AAAI97 Workshop on AI Methods in Fraud and Risk Management*, 1997.

[16] T.F. Lunt. Ides: an intelligent system for detecting intruders. In *Proceedings of the Symposium: Computer Security, Threat and Countermeasures*, November 1990. Rome, Italy.

[17] T.F. Lunt. A survey of intrusion detection techniques. *Computers and Security*, 12:405–418, 1993.

[18] T.F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection system. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, April 1988.

[19] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannthan, C. Jalali, H.S. Javitz, A. Valdos, P.G. Neumann, and T.D. Garvey. A real-time intrusion-detection expert system (ides). Technical Report, Computer Science Laboratory, SRI Internationnal, February 1992.

[20] F. Monrose and A. Rubin. Authentication via keystroke dynamics. In *4th ACM Conference on Computer and Communications Security*, April 1997.

[21] P.A. Porras and R.A. Kemmerer. Penetration state transition analysis - a rule-based intrusion detection approach. In *Eighth Annual Computer Security Applications Conference*, pages 220–229. IEEE Computer Society Press, November 1992.

[22] P.A. Porras and P.G. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, October 1997.

[23] G. Vigna and R.A. Kemmerer. Netstat: A network-based intrusion detection approach. In *Proceedings of the 1998 Annual Computer Security Applications Conference (ACSAC'98)*, pages 25–34, Los Alamitos, CA, December 1998. IEEE Computer Society, IEEE Computer Society Press. Scottsdale, AZ.

The Design of a Cryptographic Security Architecture

Peter Gutmann
University of Auckland, Auckland, New Zealand
pgut001@cs.auckland.ac.nz

Abstract

Traditional security toolkits have concentrated mostly on defining a programming interface (API) and left the internals up to individual implementors. This paper presents a design for a portable, flexible security architecture based on traditional computer security models involving a security kernel which controls access to security-relevant objects and attributes based on a configurable security policy. Layered on top of the kernel are various objects which abstract core functionality such as encryption and digital signature capabilities, certificate management, and secure sessions and data enveloping (email encryption) in a manner which allows them to be easily moved into cryptographic devices such as smart cards and crypto accelerators for extra performance or security. The versatility of the design has been proven through its use in implementations ranging from from 16-bit microcontrollers through to supercomputers, as well as a number of unusual areas such as security modules in ATM's.

1. Introduction

Traditionally, security toolkits have been implemented using a "collection of functions" design in which each encryption capability is wrapped up in its own set of functions. For example there might be a "load a DES key" function, an "encrypt with DES in CBC mode" function, a "decrypt with DES in CFB mode" function, and so on [1][2]. More sophisticated toolkits hide the plethora of algorithm-specific functions under a single set of umbrella interface functions with often complex algorithm-selection criteria, in some cases requiring the setting of up to a dozen parameters to select the mode of operation [3][4][5][6]. Either approach requires that developers tightly couple the application to the underlying encryption implementation, requiring a high degree of cryptographic awareness from developers and forcing each new algorithm and application to be treated as a distinct development.

Alternative approaches concentrate on providing functionality for a particular type of service such as authentication, integrity, or confidentiality. An example of this type of design is the GSS-API [7], which is session-oriented and is used to control session-style communications with other entities (an example implementation consists of a set of GSS-API wrapper functions for Kerberos), the OSF DCE security API [8], which is based around ACL's and secure RPC, and the SESAME API [9] which is based around a Kerberos derivative with various enhancements such as X.509 certificate support. This type of design typically includes features specific to the required functionality, in the case of the session-oriented interfaces mentioned above this is the security context which contains details of a relationship between peers based on credentials established between the peers. A non-session-based variant is the IDUP-GSS-API [10], which attempts to stretch the GSS-API to cover store-and-forward use (this would typically be used for a service such as email protection).

Both of these approaches represent an outside-in approach which begins with a particular programming interface and then bolts on whatever is required to implement the functionality in the interface. This paper presents an alternative inside-out design which takes a general crypto/security architecture and then wraps a language-independent interface around it to make particular portions of the architecture available to the user. In this case it is important to distinguish between the architecture and the API used to interface to it — with most approaches the API *is* the architecture, whereas the approach presented in this paper concentrates on the internal architecture only. Apart from the very generic APKI requirements [11], only CDSA [12] appears to provide a general architecture design, and even this is presented at a rather abstract level and defined mostly in terms of the API used to access it.

In contrast to these approaches, the design presented in this paper begins by defining the general requirements for an object model upon which to build the architecture, which is used to encapsulate various types of functionality such as encryption and certificate management. The first portion of the paper presents the overall design goals for the architecture, as well as the details of each object class. Since the entire architecture has very stringent security requirements, the object model requires an underlying security kernel capable of supporting it, one which includes a means of mediating access to objects, controlling the way this access is performed (for example the manner in which object attributes may be manipulated), and ensuring strict isolation of objects (that is, ensuring that one object can't influence the operation of another object in an uncontrolled manner). The security aspects of the architecture are covered in the second part of the paper.

2. Architecture Design Goals

An earlier work [13] gives the design requirements for a general-purpose API, including algorithm, application, and cryptomodule independence, safe programming (protection against programmer mistakes), a security perimeter to prevent sensitive data from leaking out into untrusted applications, and legacy support. Most of these requirements are pure API issues and won't be covered in any more detail here. The architecture presented here is built on the following design principles:

- Independent objects. Each object is responsible for managing its own resource requirements such as memory allocation and use of other required objects, and the interface to other objects is handled in an object-independent manner. For example a signature object would know that it is associated with a hash object, but wouldn't need to know any details of it's implementation such as function names or parameters in order to communicate with it. In addition each object has associated with it various security properties such as mandatory and discretionary ACL's, some of which are controlled for the object by the architectures security kernel and some object-specific properties which are controlled by the object itself.
- Intelligent objects. The architecture should know what to do with data and control information passed to objects, including the ability to hand it off to other objects where required. For example if a certificate object (which contains only certificate-related attributes but has no inherent encryption or signature capabilities) is asked to verify a signature using the key contained in the certificate, the architecture will hand the task off to the appropriate signature checking object without the user having to be aware that this is occurring. This leads to a very natural interface in which the user knows an object will Do The Right Thing with any data or control information sent to it, without requiring it to be accessed or used in a particular manner.
- Platform-independent design. The entire architecture should be easily portable to a wide variety of hardware types and operating systems without any significant loss of functionality. A counterexample to this design requirement is CryptoAPI 2.x [14], which is so heavily tied into features of the very newest versions of Win32 that it would be almost impossible to move to other platforms. In contrast the architecture described here was designed from the outset to be extremely portable and has been implemented on everything from 16-bit microcontrollers with no filesystem or I/O capabilities through to supercomputers, as well as unconventional designs like multiprocessor Tandem machines and IBM VM/ESA mainframes.
- Full isolation of architecture internals from external code. The architecture internals are fully decoupled from access by external code, so that the implementation may reside in its own address space (or even physically separate hardware) without the user being aware of this. The reason for this requirement is that it very clearly defines the boundaries of the architecture's trusted computing base (TCB), allowing the architecture to be defined and analysed in terms of traditional computer security models.
- Layered design. The architecture represents a true object-based multilayer design, with each layer of functionality built on its predecessor. The purpose of each layer is to provide certain services to the layer above it, shielding that layer from the details of how the service is actually implemented. Between each layer is an interface which allows data and control information to pass across in a controlled manner. In this way each layer provides a set of well-defined and understood functions which both minimise the amount of information which flows from one layer to another, and makes it easy to replace the implementation of one layer with a completely different implementation (for example migrating a software implementation into secure hardware), because all that a new layer implementation requires is that it offer the same service interface as the one it replaces.

In addition to the layer-based separation, the architecture separates individual objects within the layer into discrete, self-contained objects which are independent of other objects both within their layer and in other layers. For example in the lowest layer the basic objects typically represent an instantiation of a single encryption, digital signature, key exchange, hash, or MAC algorithm. Each object can represent a software implementation, a hardware implementation, a hybrid of the two, or some other implementation.

3. The Object Model

The architecture implements two types of objects, container objects and action objects. A container object is an object which contains one or more items such as data, keys, certificates, security state information, and security attributes. The container types can be broken down roughly into three types, data containers (referred to as envelope or session objects), key and certificate containers (keyset objects), and security attribute containers (certificate objects). An action object is an object which is used to perform an action such as encrypting, hashing, or signing data (referred to as an encryption context). Action objects are fairly simple and encapsulate the functionality of a security algorithm such as DES or RSA, these function mainly as building blocks used by the more complex object types. In addition to these standard object types, there is also a device object type which constitutes a meta-object used to work with external encryption devices such as smart cards or Fortezza cards which may require extra functions such as activation with a user PIN before they can be used. Once they're initialised as required, they can be used like any of the other object types whose functionality they provide, for example an RSA action object could be created through the device object for a smart card with RSA capabilities, or a certificate object could be stored in a device object for a Fortezza card as if it were a keyset.

The implementation of each object is completely hidden from the user, so that the only way they can access the object is by sending information to it across a carefully-controlled channel. Figure 1 illustrates how three low-level action objects (implementing DES, SHA-1, and RSA) would be handled. The object handles are small integer values, unrelated to the object itself, which are used to pass control information and data to and from the object. Since each object is referred to through an abstract handle, the interface to the object is a message-based one in which messages are sent to and received from the object. Although the external programming interface can be implemented to look like the traditional "collection of functions" one, this is simply the message-passing interface wrapped up to look like a more traditional functional interface.

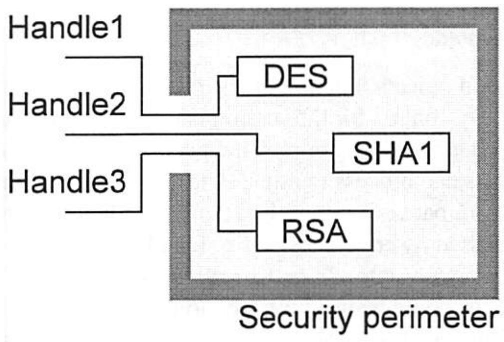


Figure 1. Typical low-level objects

Container objects generally contain other objects (as well as data and attributes) within them. For example each certificate object has an (internal) public or private key context attached to it as shown in Figure 2.

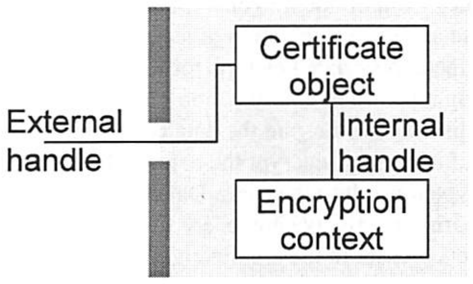


Figure 2. Object with dependent object

This encryption context can't be directly accessed by the user, but can be used in the carefully controlled manner provided by the certificate object. For example if the certificate object contains an attribute specifying that the attached public-key context may only be used for digital signature (but not encryption) purposes then any attempt to use the object for encryption purposes would be flagged as an error.

The three types of container object differ mainly in the view they present to the user, and are explained below.

3.1. Data Containers

Data containers (envelope and session objects) are objects whose behaviour is modified by the data and attributes which are pushed into them. To use an envelope, the user pushes in control information in the form of container or action objects or general attributes which control the behaviour of the container. Any data which is pushed into the container is then modified according to the behaviour established by the control information. For example if a digital signature action object was added to the data container as control information then data pushed into the container would be digitally signed; if a password attribute was pushed into the container then data pushed in would be encrypted.

Session objects function in a similar manner, but the security context for the session is usually established by exchanging information with a peer, and the session objects can process multiple data objects (for example network packets) rather than the single data object processed by envelopes — session objects are envelope objects with state. In real-world terms, envelope objects would be used for functions like S/MIME and PGP while session objects would be used for functions like SSL and ssh.

This type of object can be regarded as an intelligent container which knows how to handle data provided to it based on control information it receives. For example if the user pushes in a password attribute followed by data, the object knows that the presence of this attribute implies a requirement to encrypt data and will therefore create an encryption action object, turn the password into the appropriate key type for the object (typically through the use of a hash action object), generate an initalisation vector, pad the data out to the cipher block size if necessary, encrypt the data, and return the encrypted result to the user. Data containers, although appearing relatively simple, are by far the most complex objects present in the architecture.

3.2. Key and Certificate Containers

Key and certificate containers (keyset objects) are simple objects which contain one or more public or private keys or certificates, and may contain additional information such as key revocation data (CRL's). To the user, they appear as an (often large) collection of encryption contexts or certificate objects. Two typical container objects of this type are shown in Figure 3. Although the diagram implies the presence of huge numbers of objects, these are only instantiated when required by the user. Keyset objects are tied to whatever underlying storage mechanism is used to hold keys, typically flat files, PGP keyrings, relational databases containing certificates and CRL's, LDAP directories, or HTTP links to certificates published on web pages.

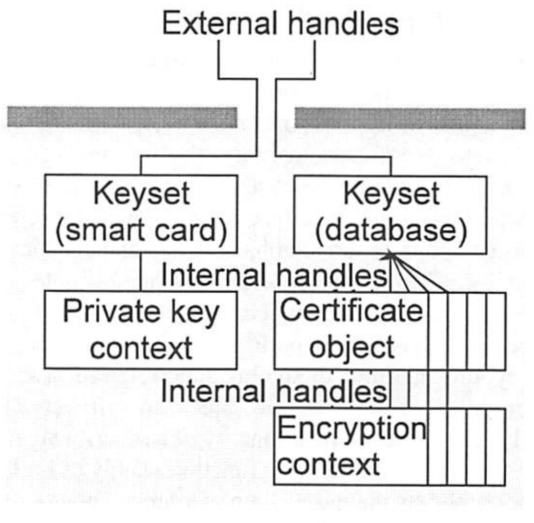


Figure 3. Container objects

3.3. Security Attribute Containers

Security attribute containers (certificate objects) contain a collection of attributes attached to a public/private key, or attributes which are attached to other information (for example signed data often comes with accompanying attributes such as the signing time and details on the signer of the data and the conditions under which the signature was generated). The most common type of security attribute container is the key certificate, which contains attribute information for a public or private key.

4. Security Features of the Architecture

Because of the lack of inter- and intra-process security present in a number of widely-deployed systems, it becomes necessary for the architecture to provide its own object security mechanisms. Security-related functions which handle sensitive data pervade the architecture, which implies that security needs to be considered in every aspect of the design, and must be designed in from the start (it's very difficult to bolt on security afterwards). Although full coverage of the various security models and requirements is beyond the scope of this paper, one typical source [15] provides a set of requirements for a security architecture which are implemented in the cryptlib security toolkit [16] in the following manner:

- Permission-based access: The default access/use permissions should be deny-all, with access or usage rights being made selectively available as required. Objects are only visible to the process which created them, although the default object access setting makes it available to every thread in the process. The reason for this is because of the requirement for ease of use — having to explicitly hand an object off to another thread within the process would significantly reduce the ease of use of the architecture. For this reason the deny-all access is made configurable by the user, with the option of making an object available throughout the process or only to one thread when it is created. If the user specifies this behaviour when the object is created then only the creating thread can see the object unless it explicitly hands off control to another thread.

- Least privilege and isolation: Each object should operate with the least privileges possible to minimise damage due to inadvertent behaviour or malicious attack, and objects should be kept logically separate in order to reduce inadvertent or deliberate compromise of the information or capabilities they contain. These two requirements go hand in hand, since each object only has access to the minimum set of resources required to perform its task, and can only use those in a carefully controlled manner. For example if a certificate object has an encryption object attached to it, the encryption object can only be used in a manner consistent with the attributes set in the certificate object (it might be usable only for signature verification, but not for encryption or key exchange, or for the generation of a new key for the object).
- Complete mediation: Each object access is checked each time the object is used — it's not possible to access an object without this checking, since the act of mapping an object handle to the object itself is synonymous with performing the access check.
- Economy of mechanism and open design: The protection system design should be as simple as possible in order to allow it to be easily checked, tested, and trusted, and should not rely on security through obscurity. To meet this requirement, the security kernel is contained in a single module, which is divided into single-purpose functions of a dozen or so lines of code which were designed and implemented using "Design by Contract" principles [17], making the kernel very amenable to testing using mechanical verifiers such as ADL [18].
- Easy to use: In order to promote its use, the protection system should be as easy to use and transparent as possible to the user. In almost all cases the user isn't even aware of the presence of the security functionality, since the programming interface can be set up to function in a manner which is almost indistinguishable from the conventional collection-of-functions interface.

A final requirement which is given is the separation of privilege, in which access to an object depends on more than one item such as a token and a password or encryption key. This is somewhat specific to user access to a computer system or objects on a computer system, and doesn't really apply to an encryption architecture.

The architecture employs a security kernel to implement its security mechanisms. This kernel provides the interface between the outside world and the architecture's objects (intra-object security) and between the objects themselves (interobject security). The security-related functions are contained in the security kernel for the following reasons [19]:

- Separation: By isolating the security mechanism from the rest of the implementation, it is easier to protect them from manipulation or penetration.
- Unity: All security functions are performed by a single code module.
- Modifiability: Changes to the security mechanism are easier to make and test.
- Compactness: Because it performs only security-related functions, the security kernel is likely to be small.
- Coverage: Every access to a protected object is checked by the kernel.

The security kernel which performs these functions is the basis of the entire architecture — all objects are accessed and controlled through it, and all object attributes are manipulated through it. The security kernel is implemented as an interface layer which sits on top of the objects, monitoring all accesses and handling all protection functions. An example of the final architectural model is shown in Figure 4, which illustrates the connection between the user application and architecture objects, with the light grey lines representing items with implicit connections (the kernel to the ACL's and one object to another, typically a certificate object to an associated encryption context.

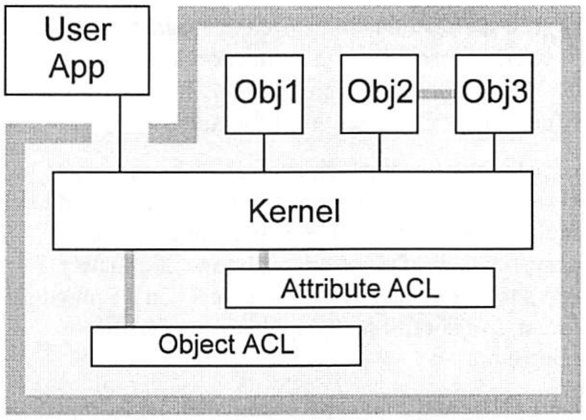


Figure 4. Architecture security model

5. Object Security and Access Control

The most important security feature of the architecture is that each object is contained entirely within its security perimeter, so that data and control information can only flow in and out in a very tightly-controlled manner, and that objects are isolated from each other within the perimeter by the security kernel. For example once keying information has been sent to an object, it can't be retrieved by the user except under tightly-controlled conditions (the only real case where this can occur is when an object's ACL permits a short-term session key to be exported in encrypted form, or a private key to be stored in encrypted form to a permanent storage medium such as a smart card or disk). In general keying information isn't even visible to the user, since it is generated inside the object itself and never leaves the security perimeter. This design is ideally matched to hardware implementations which perform strict red/black separation, since sensitive information can never leave the hardware.

Associated with each object is a mandatory access control list (ACL) which determines who can access a particular object and under which conditions the access is allowed. At a very coarse level, each object has a mandatory access control setting which determines whether it is externally visible or not (that is, whether it has a handle which is valid outside the security perimeter). Only externally visible objects can be (directly) manipulated by the user, with ACL enforcement being handled by the architectures security kernel.

Another ACL entry is used to determine which processes or threads can access an object. This entry is set by the object's owner either when it is created or at a later point when the security properties of the object are changed, and provides a much finer level of control than the internal/external access ACL. Since an object can be bound to a process or a thread within a process by an ACL, it may be invisible to other processes or threads, resulting in an access error if an attempt is made to access it from another process or thread.

A typical example of this ACL's use is shown in Figure 5, which illustrates the case of an object created by a central server thread setting up a key in the object and then handing it off to a worker thread which uses it to encrypt or decrypt data. This model is typical of multithreaded server processes which use a core server thread to manage initial connections and then hand further communications functions off to a collection of active threads.

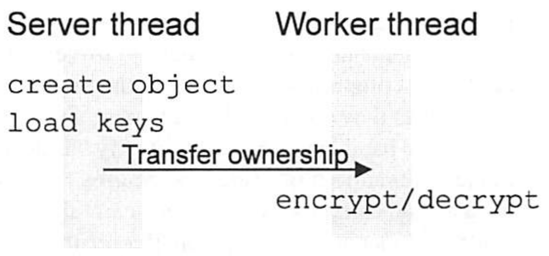


Figure 5. Object ownership transfer

Operating at a much finer level of control than the object ACL is the discretionary access control (DACL) mechanism through which only certain capabilities in an object may be enabled. For example once an encryption context is established, it can be restricted to only allow basic data encryption and decryption, but not encrypted session key export. In this way a trusted server thread can hand the context off to a worker thread without having to worry about the worker thread exporting the session key contained within it[1]. Similarly, a signature object can have a DACL set which allows it to perform only a single signature operation before it is automatically disabled by the security kernel, closing a rather troublesome security hole in which a crypto device such as a smart card can be used to authenticate arbitrary numbers of transactions by a rogue application. The usual way of implementing this fine level of control is to use a security attribute object containing the control information attached to an encryption context. Enforcement of finer-grained attribute-based DACL settings is handled by the object itself, since these settings are specific to each object type.

ACL's are inherited across objects, so that retrieving a private key encryption object from a keyset container

[1] Obviously chosen-plaintext and similar attacks are still possible, but this is something which can never be fully prevented, and which provides an attacker far less opportunity than the presence of a straight key export facility.

object will copy the container object's ACL across to the private key encryption object.

5.1. Object Security Implementation

When an object is created, it is identified to the entity which requested its creation through an arbitrary handle, an integer value which has no connection to the objects data or associated code. The handle represents an entry in an internal object table which contains information such as a pointer to the objects data and ACL information for the object. Both the object table and the object data are protected through locking and ACL mechanisms. Creating a new object works as follows:

```
caller requests object creation by kernel

lock object table;
create new object with requested type and
   attributes;
if( object was created successfully )
   add object to object table;
   set object state = under construction
unlock object table;

caller completes object-specific
   initialisation
caller sends initialisation complete
   message to kernel

lock object table
set object state = normal;
unlock object table
```

This simply creates an object of the given type with the given attributes, adds an entry for it to the object table, marks it as under construction so it can't be accessed in the incomplete state, and returns a pointer to the object data to the caller (the caller being code within the architecture itself, the user never has access to this level of functionality). At this point the caller can complete any object-specific initialisation, after which it sends an "init complete" message to the kernel which sets the objects state to normal, unlocks the object and returns its handle to the user.

The object table is maintained by the security kernel. When a new object is created, it tries to allocate a handle into the object table, with the handles being allocated in a pseudorandom manner, not so much for security purposes but to avoid the problem of the user freeing a handle by destroying an object and then immediately having the handle reused for the next object allocated, leading to problems if some of the users code still expects to find the previous object accessible through the handle. If the object table is full, it is expanded to make room for more entries. When an object is created, the kernel sets an ACL entry which marks it as being visible only within the architecture, so that the calling routine has to explicitly make it accessible outside the architecture by changing the ACL (that is, it defaults to deny-all rather than permit-all). The object can also have a variety of attributes specified for its creation such as the type of memory used (some systems can allocate limited amounts of protected, nonpageable memory which is preferred for sensitive data such as encryption contexts).

When the user passes an objects handle to cryptlib, it performs the following actions:

```
lock object table;
verify that the handle is valid;
verify that the object allows this type of
   operation;
verify that the ACL allows external access;
verify that the ACL allows access by the
   calling thread;
if( access allowed )
   set object state = processing message;
   further messages will be enqueued for
      later processing
   unlock object table
   forward message to object
   lock object table
   set object state = normal
unlock object table
```

This performs the necessary ACL checking for the object in an object-independent manner. The link from external handles through the cryptlib-wide object table and ACL check to the object itself is shown in Figure 6.

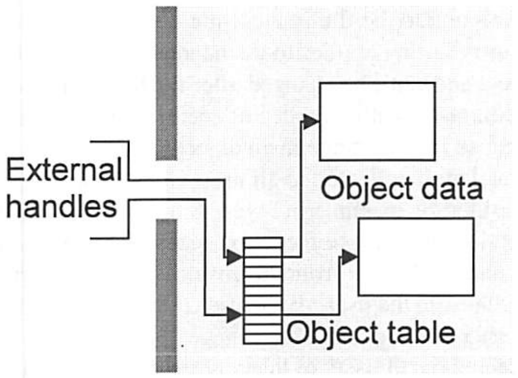


Figure 6. Object ACL checking

DACL checking is object-specific and is performed by the object itself once the basic ACL check is passed as shown in Figure 7. Architecture-internal objects are checked in a similar manner, except that the check of the external access ACL is omitted.

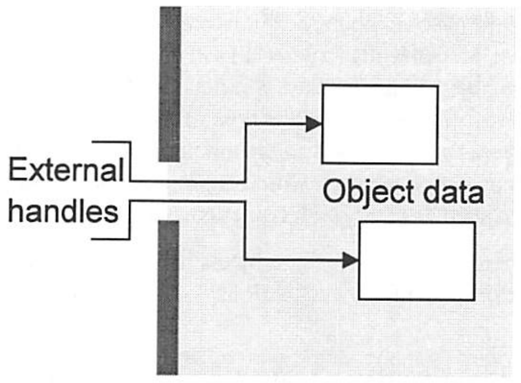


Figure 7. Object DACL checking

The ACL check is performed each time an object is used, and the ACL is attached to the object itself rather than to the handle. This means that if an ACL is changed, the change immediately affects all users of the object rather than just the owner of the handle which changed the ACL. This is in contrast to the Unix security model in which an access check is performed once when an object is instantiated (for example when a file is created or opened) and the access rights which were present at that time remain valid for the lifetime of the handle to the object. For example if a file is temporarily made world-readable and a user opens it, the handle remains valid for read access even if read permission to the file is subsequently removed — the security setting applies to the handle rather than to the object and can't be changed after the handle is created. In contrast cryptlib applies its security to the object itself, so that a change in an objects ACL is immediately reflected to all users of the object. Consider the example in Figure 8, in which an envelope contains an encryption context accessed either through the internal handle from the envelope or the external handle from the user. If the user changes the ACL for the encryption context the change is immediately reflected on all users of the context, so that any future use of the context by the envelope will result in access restrictions being enforced using the new ACL.

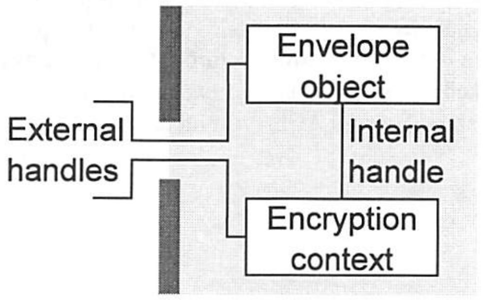


Figure 8. Objects with multiple references

Each object can be accessible to multiple threads or to a single thread. The thread access ACL is handled as part of the thread locking mechanism used to make the architecture thread-safe, and tracks the identity of the thread which owns the resource. By setting the thread access ACL, a thread can claim an unowned object, relinquish a claim to an owned object, and transfer ownership of an object to another thread. In general critical objects such as encryption contexts will be claimed by the thread which created them and will never be relinquished until the object is destroyed — to all other threads in the process the object doesn't appear to exist.

Making each object thread-safe and providing an ACL capability across multiple operating systems is somewhat tricky. The locking and ACL capabilities in cryptlib are implemented as a collection of preprocessor macros which are designed to allow them to be mapped to appropriate OS-specific user- and system-level thread synchronisation and locking functions. Great care has been taken to ensure that this locking mechanism is as fine-grained as possible, with locks typically covering no more than a dozen or so lines of code before they are relinquished, and the code executed while the lock is active being carefully scrutinised to ensure it can never become the cause of a bottleneck, for example by executing a long-running loop while the lock is active.

5.2. Object attribute security

The discussion of security features has so far concentrated on object security features, however the same security mechanisms are also applied to object attributes. An object attribute is a property belonging to an object or a class of objects, for example encryption, signature, and MAC contexts have a key attribute associated with them, certificate objects have various validity period attributes associated with them, and device objects typically have some form of PIN attribute associated with them.

Just like objects, each attribute has an ACL which specifies how it can be used and applied, with ACL enforcement being handled by the security kernel. For example the ACL for a key attribute for a triple DES encryption context would have the following entries:

```
attribute label = CRYPT_CTXINFO_KEY
type = octet string
permissions = write-once
size = 192 bits minimum, 192 bits maximum
```

In this case the ACL requires that the attribute value be exactly 192 bits long (the size of a three-key triple DES key), and it will only allow it to be written once (in other words once a key is loaded it can't be overwritten, and it can never be read). The kernel checks all data flowing in and out against the appropriate ACL, so that not only data flowing from the user into the architecture (for example identification and authentication information) but also the limited amount of data which is allowed to flow from the architecture to the user (for example status information) is carefully monitored by the kernel.

Ensuring that external software can't bypass the kernel's ACL checking requires very careful design of the I/O mechanisms to ensure that no access to architecture-internal data is ever possible. Consider the fairly typical situation in which an encrypted private key is read from disk by an application, decrypted using a user-supplied password, and used to sign or decrypt data. Using techniques such as patching the systemwide vectors for file I/O routines (which are world-writeable under Windows NT) or debugging facilities like `truss` and `ptrace` under Unix, hostile code can determine the location of the buffer into which the encrypted key is copied and monitor the buffer contents until they change due to the key being decrypted, at which point it has the raw private key available to it. An even more serious situation occurs when a function interacts with untrusted external code by supplying a pointer to information located in an internal data structure, in which case an attacker can take the returned pointer and add or subtract whatever offset is necessary to read or write other information which is stored nearby. With a number of current security toolkits, something as simple as flipping a single bit is enough to turn off some of the encryption (and in at least one case turn on much stronger encryption than the US-exportable version of the toolkit is supposed to be capable of), cause keys to be leaked, and have a number of other interesting effects.

In order to avoid these problems, the architecture never provides direct access to any internal information. All object attribute data is copied in and out of memory locations supplied by the external software into separate (and unknown to the external software) internal memory locations. In cases where supplying pointers to memory is unavoidable (for example where it's required for `fread` or `fwrite`), the supplied buffers are scratch buffers which are decoupled from the architecture-internal storage space in which the data will eventually be processed.

This complete decoupling of data passing in or out means that it is very easy to run an implementation of the architecture in its own address space or even in physically separate hardware without the user ever being aware that this is the case, for example under Unix the implementation would run as a daemon owned by a different user and under Windows NT it would run as a system service. Alternatively, the implementation can run on dedicated hardware which is physically isolated from the host system.

5.3. Benefits of the object security model

This particular object security model has several advantages. By combining the locking which is required to make the objects thread-safe with part of the ACL functionality, very little extra overhead is introduced (of the two most common operating systems with threading capabilities, both Unix user-level pthreads implementations and Windows pseudocritical sections don't usually require any kernel calls, making them relatively quick). In addition since each object is inherently thread-safe and ACL-protected, different parts of cryptlib can use and communicate with the objects without having to worry about access checking and problems with simultaneous access to shared resources — the kernel and objects take care of this themselves. Since attribute checking is also performed using ACL's rather than the traditional ad hoc parameter checks hardcoded into miscellaneous functions in various places, all attributes can have a coordinated security policy applied to them through a central, easily-checked set of ACL's.

6. Object Internals

Creating or instantiating a new object involves obtaining a new handle, allocating and initialising an internal data structure which stores information on the object, setting ACL's, connecting the object to any underlying hardware or software if necessary (for example establishing a session with a smart card reader or database backend), and finally returning the object's handle to the user. Although the user sees a single object type which is consistent across all computer systems and implementations, the exact (internal) representation of the object can vary considerably. In the simplest case, an object consists of a thin mapping layer which translates calls from the architectures's internal API to the API used by a hardware implementation. Since encryption contexts, which represent the lowest level in the architecture, have been designed to map directly onto the functionality provided by common hardware crypto accelerators, these can be used directly when appropriate hardware is present in the system.

If the encryption hardware consists of a crypto device with a higher level of functionality or even a general-purpose secure coprocessor rather than just a simple crypto accelerator, more of the functionality can be offloaded onto the device or secure coprocessor. For example while a straight crypto accelerator may support functionality equivalent to basic DES and RSA operations on data blocks, a crypto device such as a PKCS #11 token would provide extended functionality including the necessary data formatting and padding operations required to perform secure and portable key exchange and signature operations, and more sophisticated secure coprocessors which are effectively scaled-down PC's [20] can take on board architecture functionality at an even higher level. Figure 9 shows the levels at which external hardware functionality can be integrated, with the lowest level corresponding to the functionality embodied in an encryption context, while the higher levels correspond to functionality in envelope and certificate objects.

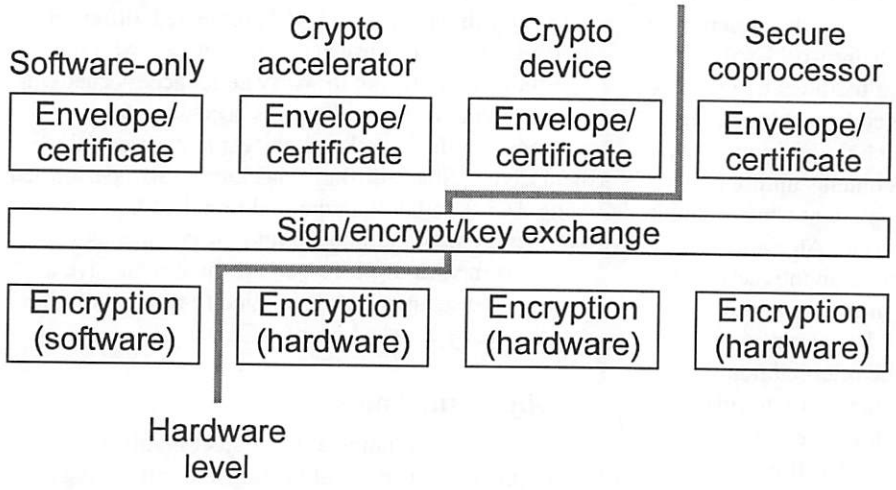


Figure 9. Mapping of architecture functionality levels to crypto/security hardware

6.1. Object Internal Details

Although each type of object differs considerably in its internal design, they all share a number of common features which will be covered here. Each object consists of three main parts:

1. State information, stored either in secure or general-purpose memory depending on its sensitivity.
2. The object's message handler.
3. A set of function pointers for the methods used by the object.

The actual functionality of the object is implemented through the function pointers, which are initialised when the object is instantiated to refer to the appropriate methods for the object. Using an instantiation of a DES encryption context with an underlying software implementation and an RSA encryption context with an underlying hardware implementation, we have the encryption context structures shown in Figure 10.

When the two objects are created, the DES context is plugged into the software DES implementation and the RSA context is plugged into a hardware RSA accelerator. Although the low-level implementations are very different, both are accessed through the same methods, typically `context.loadKey()`, `context.encrypt()`, and `context.decrypt()`. Substituting a different implementation of an encryption algorithm (or adding an entirely new algorithm) requires little more than creating the appropriate interface methods to allow a context to be plugged into the underlying implementation. As an example of how simple this can be, when the Skipjack algorithm was declassified [21] it took only a few minutes to plug in an implementation of the algorithm, providing full support for Skipjack throughout the entire architecture and to all applications which employed the architecture's standard capability query mechanism, which automatically establishes the available capabilities of the architecture on startup.

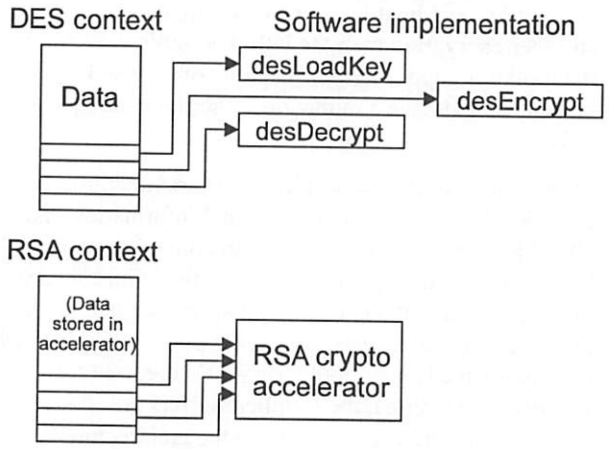


Figure 10. Encryption context internal structure

6.2. Object Reuse

Since object handles are detached from the objects they are associated with, a single object can (provided its ACL's allow this) be used by multiple processes or threads at once. This flexibility is particularly important with objects used in connection with container objects, since replicating every object pushed into a container creates both unnecessary overhead and increases the chances of compromise of sensitive information if keys and other data are copied across to each newly created object.

Instead of copying each object whenever it is reused, the architecture maintains a reference count for it and only copies it when necessary. In general the copying is only needed for some encryption contexts, which employ a copy-on-write mechanism which ensure the object isn't replicated unnecessarily. Other objects which can't (easily) be replicated, or which don't need to be replicated, have their reference count incremented when they are reused, and decremented when they are freed. The object itself is only destroyed when its reference count drops to zero.

To see how this works, let's assume the user creates an encryption context and pushes it into an envelope object. This results in a context with a reference count of 2, with one external reference (by the user) and one internal reference (by the envelope object) as shown previously in Figure 8. Typically the user would then destroy the encryption context while continuing to use the envelope which it is now associated with. The reference with the external access ACL would be destroyed and the reference count decremented by one, leaving the object as shown in Figure 11 with a reference count of 1 and an internal access ACL.

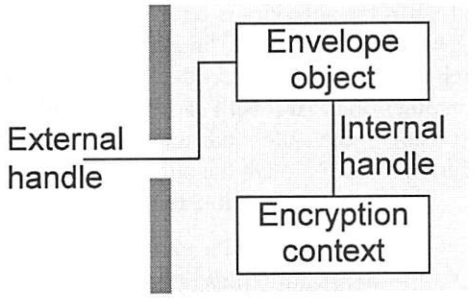


Figure 11. Objects with multiple references after the external reference is destroyed

To the user the object has indeed been destroyed, since it is now accessible only to the envelope object. When the envelope object is destroyed the encryption context's reference count is again decremented through a message sent from the envelope, leaving it at zero whereupon the kernel sends it a "destroy object" message to notify it to shut itself down, after which it removes the object from the object table. The only time objects are explicitly destroyed is through an external signal such as a smart card withdrawal or when the kernel broadcasts destroy object messages when it is closing down. At any other time only their reference count is decremented.

In some cases an object has to change its behaviour when it is reused. For example some key databases don't handle multiple independent writes at all well (this can upset their internal buffering and state management), so the object will change its ACL from read/write to read-only when its reference count becomes greater than one. Once it drops back to one, the ACL is updated to allow read/write access again.

The use of the reference-counting implementation allows objects to be treated in a far more flexible manner than would otherwise be the case. For example the paradigm of pushing attributes and objects into envelopes (which could otherwise be prohibitively expensive due to the overhead of making a new copy of the object for each envelope) is rendered feasible since in general only a single copy of each object exists. Similarly, a single (heavyweight) connection to a key database can be shared across multiple threads or processes, an important factor in a number of client/server databases where a single client connection can consume a megabyte or more of memory.

6.3. Data Formats

Since each object represents an abstract security concept, none of them are tied to a particular underlying data format or type. For example an envelope could output the result of its processing in the data format used by CMS/S/MIME, PGP, PEM, MSP, or any other format required. As with the other object types, when the envelope object is created, its function pointers are set to encoding or decoding methods which handle the appropriate data formats. In addition to the variable, data format-specific processing functions, envelope and certificate objects employ data recognition routines which will automatically determine the format of input data (for example whether data is in CMS/S/MIME or PGP format, or whether a certificate is a certificate request, certificate, PKCS #7 certificate chain, CRL, or other type of data) and set up the correct processing methods as appropriate.

7. Interobject Communications

Objects communicate internally via a message-passing mechanism, although this is typically hidden from the user by a more conventional functional interface. The message-passing mechanism connects the objects indirectly, replacing pointers and direct function calls and is the fundamental mechanism used to implement the complete isolation of architecture internals from the outside world. Since the mechanism is anonymous, it reveals nothing about an objects implementation or its interface, or even its existence. The message-passing mechanism has three parts, the source object, the destination object, and the message dispatcher. In order to send a message from a source to a destination, the source object needs to know the target objects handle, but the target object has no knowledge of where a message came from unless the source explicitly informs it of this. All data communicated between the two is held in the message itself.

To handle interobject messaging, the kernel contains a message dispatcher which maintains an internal message queue used to forward messages to the appropriate object or objects. Some messages are directed at a particular object (identified by the objects handle), others to an entire class of object or even to all objects. For example if an encryption context is instantiated from a smart card and the card is then withdrawn from the reader, the event handler for the keyset object associated with the reader broadcasts a card withdrawal message identifying the card which was removed to all active objects as illustrated in Figure 12. This is necessary to notify the encryption context that it may need to take action based on the card withdrawal, and also to notify further objects such as envelope objects and certificates which have been created or acted upon by the encryption context — since the sender is completely disconnected from the receiver, it needs to broadcast the message to all objects to ensure that everything which might have an interest is notified. The message handler has been designed so that processing a message of this type has almost zero overhead compared to the complexity of tracking which message might apply to which objects, so it makes more sense to handle the notification as a broadcast rather than maintaining per-object lists of messages the object is interested in.

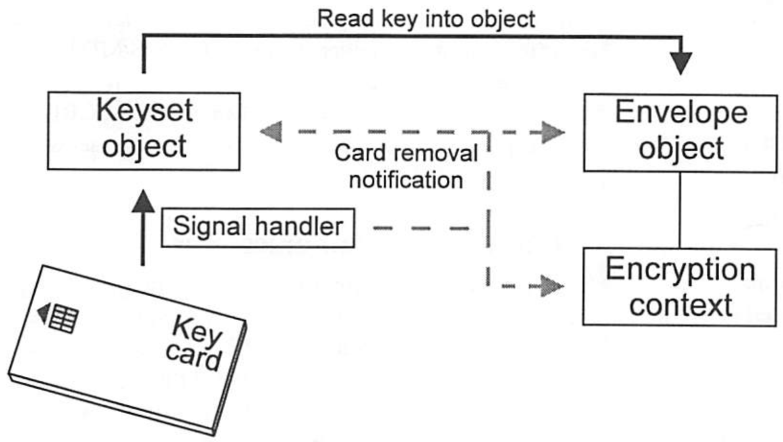


Figure 12. Interobject messaging example

As each object receives a message, it can explicitly choose to act on it or to take the default action of ignoring it. Each object therefore has the ability to intelligently handle external events in a controlled manner, processing them as appropriate. Because an object controls how it handles these events, there's no need for any other object or control routine to know about the internal details or function of the object — it simply posts a notification of an event and goes about its business.

As an example of how intelligent message fowarding across objects can work, consider an attempt to encrypt data using a certificate (in fact it's really encrypted using the encryption context associated with the certificate, but this is invisible to the user, all they see is the certificate object). This fails in some way and returns a "public-key encryption operation failed" error to the caller, who tries to get more information about what went wrong by reading the objects error information attributes. The kernel sends the certificate object a query message, but it isn't involved with the encryption operation and returns the query to the kernel with a "not at this address, try attached objects" comment, to which the kernel forwards the message to the attached encryption context. The context in turn doesn't know the exact details and returns the message to the kernel, which determines that the context is tied to some sort of underlying encryption device, so it forwards the message to the device object. The buck finally stops at the device object (which happens to be tied to a smart card which has just failed in some way), which returns the appropriate error information from the card. This demonstrates the "intelligent object" design in which an object can instruct the kernel to hand a task off to another object if it isn't capable of fulfilling the request itself — the caller ends up with telemetry from the smart card even though the object they're explicitly using is a certificate.

In the case of the card withdrawal notification illustrated in Figure 12, the affected objects which don't choose to ignore it would typically erase any security-related information, close active OS services such as open file handles, free allocated memory, and place themselves in a signalled state in which no further use of the object is possible apart from destroying it. Message queueing and dispatching is handled by the kernel's message dispatcher and the message handlers built into each object, which remove from the user the need to check for various special-case conditions such as smart card withdrawals. In practice the only object which would process the message is the encryption context. Other objects which might contain the context (for example an envelope or certificate object) will only notice the card withdrawal if they try to use the context, at which point it will inform them that it has been signalled and is no longer useable.

Since the objects act independently, the fact that one object has changed state doesn't affect any of the other objects. This object independence is an important feature, since it doesn't tie the functioning of one object to every component object it contains or uses — a smart card-based private key might only be needed to decrypt a session key at the start of a communications session, after which its presence is irrelevant. Since each object manages its own state, the fact that the encryption context created from the key on the card has become signalled doesn't matter to the object using it after it has recovered the session key.

7.1. Asynchronous vs Synchronous Message Dispatching

When processing messages, the dispatcher can handle them in one of two ways:

1. Asynchronously, returning control to the caller immediately while processing the object in a separate thread
2. Synchronously, suspending the caller while the message is processed.

There are two types of messages which can be sent to an object, simple notifications and data communications which are processed immediately, and more complex, generally object-specific messages which can take awhile to process, an example being "generate a key" which can take awhile for many public-key algorithms. This would in theory require both types of message dispatching, however in the cryptlib architecture each object is responsible for its own handling of asynchronous processing. In practice this means that (on systems which support it) the object has one or more threads attached to it which perform asynchronous processing (on non-threaded systems there's no choice but to use synchronous messaging). When a source object sends a message to a destination which may take some time to generate a result, it includes in the message its own handle to which the destination sends a response when it's ready. When the destination object has the data required by the source ready, it sends a message back to the source object containing the data. Since the objects are inherently thread-safe, the messaging mechanism is also safe when asynchronous processing is taking place.

For an example of how the handling of messaging in an asynchronous manner works, consider a status query being sent to a keyset object connected to a keyset with a slow response time (for example a smart card or a remote LDAP server). First, the source object would send a status query message, including in the message its object handle (the return address for the query result). The dispatcher would forward the message to the keyset object, which would remember the return address and return control to the dispatcher, which would in turn return control to the source object. Some time later (whenever the keyset object has the requested information), it would send a message back to the source object containing the information it had requested.

7.2. The Message Dispatcher

The message dispatcher maintains a queue of all pending messages due to be sent to target objects which are dispatched in order of arrival. If an object isn't busy processing an existing message, a new message intended for it is immediately dispatched to it without being enqueued, which prevents the single message queue from becoming a bottleneck. For group messages (messages sent to all objects of a given type) or broadcast messages (messages sent to all objects), the message is sent to every applicable object in turn.

Recursive messages (ones which result in further messages being sent to the source object) are handled by having the dispatcher enqueue messages intended for an object which already has a message present in the queue and return immediately to the caller. This ensures that the new message isn't processed until the earlier message(s) for the object have been processed. If the message is for a different object, it is either processed immediately if the object isn't already processing a message or it is prepended to the queue and processed before other messages, so that messages sent by objects to subordinate objects are processed before messages for the objects themselves. An object won't have a new message dispatched to it until the current one has been processed. This processing order ensures that messages to the same object are processed in the order sent, and messages to different objects arising from the message to the original object are processed before the message for the original object completes.

Since an earlier message can result in an object being destroyed, the dispatcher also checks to see whether the object still exists in an active state. If not, it dequeues all further messages without calling the objects message handler.

8. Other Kernel Mechanisms

In order to work with the objects described so far, the architecture requires a number of other mechanisms to handle synchronisation, background processing, and the reporting of events within the architecture to the user. These mechanisms are described below.

8.1. Semaphores

In the message-passing example given earlier, the source object may want to wait until the data it requested becomes available. In general since each object can potentially operate asynchronously, cryptlib requires some form of synchronisation mechanism which allows an object to wait for a certain event before it continues processing. The synchronisation is implemented using lightweight internal semaphores, which are used in most cases (in which no actual waiting is necessary) before falling back to the often heavyweight OS semaphores.

cryptlib provides two types of semaphores, system semaphores (that is, predefined semaphore handles corresponding to fixed resources or operations such as binding to various types of drivers which takes place on startup) and user semaphores, which are allocated by an object as required. System semaphores have architecture-wide unique handles akin to the stdio libraries predefined stdin, stdout, and stderr handles.

8.2. Threads

The independent, asynchronous nature of the objects in the architecture means that, in the worst case, there can be dozens of threads all whirring away inside cryptlib, most of which will be blocked while waiting on external events. Since this acts as a drain on system resources, can negatively affect performance (some operating systems can take some time to instantiate a new thread), and adds extra implementation detail for handling each thread, cryptlib provides an internal service thread which can be used by objects to perform basic housekeeping tasks. Each object can register service functions with this thread which are called in a round-robin fashion, after which the thread goes to sleep for a preset time interval, behaving much like a fiber or lightweight, user-scheduled thread. This means that simple tasks such as basic status checks can be performed by a single architecture-wide thread instead of requiring one thread per object. This service thread also performs general tasks such as touching each allocated memory page which is marked as containing sensitive data whenever it runs in order to reduce the chances of the page being swapped out.

Consider an example of a smart card keyset object which needs to check the card status every now and then to determine whether the card has been removed from the reader. Most serial-port based readers don't provide any useful notification mechanism, but only report a "card removed" status on the next attempt to access it. This isn't terribly useful to the architecture, which expects to be able to destroy objects which depend on the card as soon as it is removed.

In order to check for card removal, the keyset object registers a service function with the service thread. The registration returns a unique service ID which can be used later to deregister it. Deregistration can also occur automatically when the object which registered the service function is destroyed.

Once a service function is registered, it is called whenever the service thread runs. In the case of the keyset object it would query the reader to determine whether the card was still present. If the card is removed, it sends a message to the keyset object (running in a different thread), after which it returns, and the next service function is processed. In the meantime the keyset object notifies all dependent objects and destroys itself, in the process deregistering the service function. As with the message processing, since the objects involved are all thread-safe, there are no problems with synchronisation (for example the service function being called can deregister itself without any problems).

8.3. Event Notification

A common method for notifying the user of events is to use one or more callback functions. These functions are registered with a program and are called when certain events occur. Typical implementations use either event-specific callbacks (so the user can register functions only for events they're specifically interested in) or umbrella callbacks which get passed all events, with the user determining whether they want to act on them or not.

Callbacks have two main problems. The first of these is that they are inherently language and often OS-specific, often occurring across process boundaries and always requiring special handling to set up the appropriate stack frames, ensure arguments are passed in a consistent manner, and so on. Language-specific alternatives to callbacks such as Visual Basic event handlers are even more problematic.

The second problem with callbacks is that the called user code is given the full privileges of the calling code unless special steps are taken [22]. One possible workaround is to perform callbacks from a special no-privileges thread, but this means that the called code is given too few privileges rather than too many.

A better solution which avoids both the portability and security problems of callbacks is to avoid them altogether in favour of an object polling mechanism.

Since all encryption functionality is provided in terms of objects, object status checking is provided automatically by the security kernels reference monitor — if any object has an abnormal status associated with it (for example it might be busy performing a long-running operation such as a key generation), any attempt to use it wil result in the status being returned without any action being taken.

Because of the object-based approach used for all security functionality, the object status mechanism works transparently across arbitrarily linked objects. For example if the encryption object in which the key is being generated is pushed into an envelope, any attempt to use it before the key generation has completed will result in an "object busy" status being passed back up to the user. Since it's the encryption object which is busy (rather than the envelope), it's still possible to use the envelope for non-encryption functions while the key generation is occurring in the encryption object.

9. Conclusion

This paper has presented a flexible, platform-independent cryptographic security architecture which is suited to software, hardware, and hybrid implementations. By encapsulating the functionality inside independent intelligent objects protected by a central security kernel, portions of the architecture can be replaced or updated with a minimum of effort while guaranteeing a consistent interface and handling of the objects within the architecture. As implemented in cryptlib, this design has been successfully deployed on systems ranging from 16-bit microcontrollers through to supercomputers, languages ranging from C/C++ through to Perl and Visual Basic, and interfaced to a wide variety of cryptographic hardware and other devices, providing a single consistent interface across all of these platforms and languages (write once, encrypt anywhere).

10. Acknowledgements

The author would like to thank Peter Fenwick, Trent Jaeger, Paul Karger, and the referees for feedback and comments, and would also like to acknowledge Microsoft, whose operating system security motivated many of the design features of the architecture presented in this paper.

11. References

[1] libdes, `http://www.cryptsoft.com/ssleay/faq.html`, 1996.

[2] "Fortezza Cryptologic Programmers Guide", Version 1.52, National Security Agency Workstation Security Products, 30 January 1996.

[3] "BSAFE Library Reference Manual", Version 4.0, RSA Data Security, 1998.

[4] "Generic Cryptographic Service API (GCS-API)", Open Group Preliminary Specification, June 1996.

[5] "Microsoft CryptoAPI Application Programmers Guide", Version 1, Microsoft Corporation, 16 August 1996.

[6] "PKCS #11 Cryptographic Token Interface Standard", Version 2.01, RSA Laboratories, 22 December 1997.

[7] "Generic Security Service Application Programming Interface", RFC 2078, John Linn, January 1997.

[8] "DCE Security Programming", Wei Hu, O'Reilly and Associates, 1995.

[9] "SESAME Technology Version 4", December 1995 (newer versions exist but are no longer publicly available).

[10] "Independent Data Unit Protection Generic Security Service Application Program Interface (IDUP-GSS-API)", RFC 2479, Carlisle Adams, December 1998.

[11] "Architecture for Public-key Infrastructure (APKI), Draft 3", The Open Group, 27 March 1998.

[12] "Common Data Security Architecture (CDSA) Version 2.0", The Open Group, May 1999.

[13] "Security Service API: Cryptographic API Recommendation, Updated and Abridged Edition", NSA Cross Organization CAPI Team, 25 July 1997.

[14] "Microsoft Cryptographic Application Programming Interface (CryptoAPI)", Version 2, Microsoft Corporation, 22 December 1998.

[15] "The Protection of Information in Computer Systems", Jerome Saltzer and Michael Schroeder, Proceedings of the IEEE, **Vol.63, No.9** (September 1975), p.1278.

[16] cryptlib version 3, `http://www.cs.auckland.ac.nz/~pgut001/cryptlib.html`, 1999.

[17] "Object-Oriented Software Construction, Second Edition", Bertrand Meyer, Prentice Hall, 1997.

[18] "Assertion Definition Language (ADL) 2.0", X/Open Group, November 1998.

[19] "Security in Computing", Charles Pfleeger, Prentice-Hall, 1989.

[20] "Building a High-Performance Programmable, Secure Coprocessor", Sean Smith and Steve Weingart, Computer Networks and ISDN Systems, **Issue 31** (April 1999), p.831.

[21] "SKIPJACK and KEA Algorithm Specification", Version 2.0, NSA, 29 May 1998.

[22] "Java Security Architecture", JDK 1.2, Sun Microsystems Corp, 1997.

Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0

Alma Whitten
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
alma@cs.cmu.edu

J. D. Tygar[1]
EECS and SIMS
University of California
Berkeley, CA 94720
tygar@cs.berkeley.edu

Abstract

User errors cause or contribute to most computer security failures, yet user interfaces for security still tend to be clumsy, confusing, or near-nonexistent. Is this simply due to a failure to apply standard user interface design techniques to security? We argue that, on the contrary, effective security requires a different usability standard, and that it will not be achieved through the user interface design techniques appropriate to other types of consumer software.

To test this hypothesis, we performed a case study of a security program which does have a good user interface by general standards: PGP 5.0. Our case study used a cognitive walkthrough analysis together with a laboratory user test to evaluate whether PGP 5.0 can be successfully used by cryptography novices to achieve effective electronic mail security. The analysis found a number of user interface design flaws that may contribute to security failures, and the user test demonstrated that when our test participants were given 90 minutes in which to sign and encrypt a message using PGP 5.0, the majority of them were unable to do so successfully.

We conclude that PGP 5.0 is not usable enough to provide effective security for most computer users, despite its attractive graphical user interface, supporting our hypothesis that user interface design for effective security remains an open problem. We close with a brief description of our continuing work on the development and application of user interface design principles and techniques for security.

1 Introduction

Security mechanisms are only effective when used correctly. Strong cryptography, provably correct protocols, and bug-free code will not provide security if the people who use the software forget to click on the encrypt button when they need privacy, give up on a communication protocol because they are too confused about which cryptographic keys they need to use, or accidentally configure their access control mechanisms to make their private data world-readable. Problems such as these are already quite serious: at least one researcher [2] has claimed that configuration errors are the probable cause of more than 90% of all computer security failures. Since average citizens are now increasingly encouraged to make use of networked computers for private transactions, the need to make security manageable for even untrained users has become critical [4, 9].

This is inescapably a user interface design problem. Legal remedies, increased automation, and user training provide only limited solutions. Individual users may not have the resources to pursue an attacker legally, and may not even realize that an attack took place. Automation may work for securing a communications channel, but not for setting access control policy when a user wants to share some files and not others. Employees can be required to attend training sessions, but home computer users cannot.

Why, then, is there such a lack of good user interface design for security? Are existing general user interface design principles adequate for security? To answer these questions, we must first understand what kind of usability security requires in order to be

[1] Also at Computer Science Department, Carnegie Mellon University (on leave).

effective. In this paper, we offer a specific definition of usability for security, and identify several significant properties of security as a problem domain for user interface design. The design priorities required to achieve usable security, and the challenges posed by the properties we discuss, are significantly different from those of general consumer software. We therefore suspect that making security usable will require the development of domain-specific user interface design principles and techniques.

To investigate further, we looked to existing software to find a program that was representative of the best current user interface design for security, an exemplar of general user interface design as applied to security software. By performing a detailed case study of the usability of such a program, focusing on the impact of usability issues on the effectiveness of the security the program provides, we were able to get valuable results on several fronts. First, our case study serves as a test of our hypothesis that user interface design standards appropriate for general consumer software are not sufficient for security. Second, good usability evaluation for security is itself something of an open problem, and our case study discusses and demonstrates the evaluation techniques that we found to be most appropriate. Third, our case study provides real data on which to base our priorities and insights for research into better user interface design solutions, both for the specific program in question and for the domain of security in general.

We chose PGP 5.0[2] [5, 14] as the best candidate subject for our case study. Its user interface appears to be reasonably well designed by general consumer software standards, and its marketing literature [13] indicates that effort was put into the design, stating that the "significantly improved graphical user interface makes complex mathematical cryptography accessible for novice computer users." Furthermore, since public key management is an important component of many security systems being proposed and developed today, the problem of how to make the functionality in PGP usable enough to be effective is widely relevant.

We began by deriving a specific usability standard for PGP from our general usability standard for security. In evaluating PGP 5.0's usability against that standard, we chose to employ two separate evaluation methods: a direct analysis technique called cognitive walkthrough [17], and a laboratory user test [15]. The two methods have complementary strengths and weaknesses. User testing produces more objective results, but is necessarily limited in scope; direct analysis can consider a wider range of possibilities and factors, but is inherently subjective. The sum of the two methods produces a more exhaustive evaluation than either could alone.

We present a point by point discussion of the results of our direct analysis, followed by a brief description of our user test's purpose, design, and participants, and then a compact discussion of the user test results. A more detailed presentation of this material, including user test transcript summaries, may be found in [18].

Based on the results of our evaluation, we conclude that PGP 5.0's user interface does not come even reasonably close to achieving our usability standard – it does not make public key encryption of electronic mail manageable for average computer users. This, along with much of the detail from our evaluation results, supports our hypothesis that security-specific user interface design principles and techniques are needed. In our continuing work, we are using our usability standard for security, the observations made in our direct analysis, and the detailed findings from our user test as a basis from which to develop and apply appropriate design principles and techniques.

2 Understanding the problem

2.1 Defining usability for security

Usability necessarily has different meanings in different contexts. For some, efficiency may be a priority, for others, learnability, for still others, flexibility. In a security context, our priorities must be whatever is needed in order for the security to be used effectively. We capture that set of priorities in the definition below.

Definition: Security software is usable if the people who are expected to use it:

1. **are reliably made aware of the security tasks they need to perform;**
2. **are able to figure out how to successfully perform those tasks;**
3. **don't make dangerous errors; and**
4. **are sufficiently comfortable with the interface to continue using it.**

[2] At the time of this writing, PGP 6.0 has recently been released. Some points raised in our case study may not apply to this newer version; however, this does not significantly diminish the value of PGP 5.0 as a subject for usability analysis. Also, our evaluation was performed using the Apple Macintosh version, but the user interface issues we address are not specific to a particular operating system and are equally applicable to UNIX or Windows security software.

2.2 Problematic properties of security

Security has some inherent properties that make it a difficult problem domain for user interface design. Design strategies for creating usable security will need to take these properties explicitly into account, and generalized user interface design does not do so. We describe five such properties here; it is possible that there are others that we have not yet identified.

1. The unmotivated user property

Security is usually a secondary goal. People do not generally sit down at their computers wanting to manage their security; rather, they want to send email, browse web pages, or download software, and they want security in place to protect them while they do those things. It is easy for people to put off learning about security, or to optimistically assume that their security is working, while they focus on their primary goals. Designers of user interfaces for security should not assume that users will be motivated to read manuals or to go looking for security controls that are designed to be unobtrusive. Furthermore, if security is too difficult or annoying, users may give up on it altogether.

2. The abstraction property

Computer security management often involves security policies, which are systems of abstract rules for deciding whether to grant accesses to resources. The creation and management of such rules is an activity that programmers take for granted, but which may be alien and unintuitive to many members of the wider user population. User interface design for security will need to take this into account.

3. The lack of feedback property

The need to prevent dangerous errors makes it imperative to provide good feedback to the user, but providing good feedback for security management is a difficult problem. The state of a security configuration is usually complex, and attempts to summarize it are not adequate. Furthermore, the correct security configuration is the one which does what the user "really wants", and since only the user knows what that is, it is hard for security software to perform much useful error checking.

4. The barn door property

The proverb about the futility of locking the barn door after the horse is gone is descriptive of an important property of computer security: once a secret has been left accidentally unprotected, even for a short time, there is no way to be sure that it has not already been read by an attacker. Because of this, user interface design for security needs to place a very high priority on making sure users understand their security well enough to keep from making potentially high-cost mistakes.

5. The weakest link property

It is well known that the security of a networked computer is only as strong as its weakest component. If a cracker can exploit a single error, the game is up. This means that users need to be guided to attend to all aspects of their security, not left to proceed through random exploration as they might with a word processor or a spreadsheet.

2.3 A usability standard for PGP

People who use email to communicate over the Internet need security software that allows them to do so with privacy and authentication. The documentation and marketing literature for PGP presents it as a tool intended for that use by this large, diverse group of people, the majority of whom are not computer professionals. Referring back to our general definition of usability for security, we derived the following question on which to focus our evaluation:

If an average user of email feels the need for privacy and authentication, and acquires PGP with that purpose in mind, will PGP's current design allow that person to realize what needs to be done, figure out how to do it, and avoid dangerous errors, without becoming so frustrated that he or she decides to give up on using PGP after all?

Stating the question in more detail, we want to know whether that person will, at minimum:

- understand that privacy is achieved by encryption, and figure out how to encrypt email and how to decrypt email received from other people;
- understand that authentication is achieved through digital signatures, and figure out how

to sign email and how to verify signatures on email from other people;

- understand that in order to sign email and allow other people to send them encrypted email a key pair must be generated, and figure out how to do so;
- understand that in order to allow other people to verify their signature and to send them encrypted email, they must publish their public key, and figure out some way to do so;
- understand that in order to verify signatures on email from other people and send encrypted email to other people, they must acquire those people's public keys, and figure out some way to do so;
- manage to avoid such dangerous errors as accidentally failing to encrypt, trusting the wrong public keys, failing to back up their private keys, and forgetting their pass phrases; and
- be able to succeed at all of the above within a few hours of reasonably motivated effort.

This is a minimal list of items that are essential to correct use of PGP. It does not include such important tasks as having other people sign the public key, signing other people's public keys, revoking the public key and publicizing the revocation, or evaluating the authenticity of a public key based on accompanying signatures and making use of PGP's built-in mechanisms for such evaluation.

3 Evaluation methods

We chose to evaluate PGP's usability through two methods: an informal cognitive walkthrough [17] in which we reviewed PGP's user interface directly and noted aspects of its design that failed to meet the usability standard described in Section 2.3; and a user test [15] performed in a laboratory with test participants selected to be reasonably representative of the general population of email users. The strengths and weaknesses inherent in each of the two methods made them useful in quite different ways, and it was more realistic for us to view them as complementary evaluation strategies [7] than to attempt to use the laboratory test to directly verify the points raised by the cognitive walkthrough.

Cognitive walkthrough is a usability evaluation technique modeled after the software engineering practice of code walkthroughs. To perform a cognitive walkthrough, the evaluators step through the use of the software as if they were novice users, attempting to mentally simulate what they think the novices' understanding of the software would be at each point, and looking for probable errors and areas of confusion. As an evaluation tool, cognitive walkthrough tends to focus on the learnability of the user interface (as opposed to, say, the efficiency), and as such it is an appropriate tool for evaluating the usability of security.

Although our analysis is most accurately described as a cognitive walkthough, it also incorporated aspects of another technique, heuristic evaluation [11]. In this technique, the user interface is evaluated against a specific list of high-priority usability principles; our list of principles is comprised by our definition of usability for security as given in Section 2.1 and its restatement specifically for PGP in Section 2.3. Heuristic evaluation is ideally performed by people who are "double experts," highly familiar with both the application domain and with usability techniques and requirements (including an understanding of the skills, mindset and background of the people who are expected to use the software). Our evaluation draws on our experience as security researchers and on additional background in training and tutoring novice computer users, as well as in theater, anthropology and psychology.

Some of the same properties that make the design of usable security a difficult and specialized problem also make testing the usability of security a challenging task. To conduct a user test, we must ask the participants to use the software to perform some task that will include the use of the security. If, however, we prompt them to perform a security task directly, when in real life they might have had no awareness of that task, then we have failed to test whether the software is designed well enough to give them that awareness when they need it. Furthermore, to test whether they are able to figure out how to use the security when they want it, we must make sure that the test scenario gives them some secret that they consider worth protecting, comparable to the value we expect them to place on their own secrets in the real world. Designing tests that take these requirements adequately into account is something that must be done carefully, and with the exception of some work on testing the effectiveness of warning labels [19], we have found little existing material on user testing that addresses similar concerns.

4 Cognitive walkthrough

Since this paper is intended for a security audience, and is subject to space limitations, we present the results of our cognitive walkthrough in summary form, focusing on the points which are most relevant to security risks.

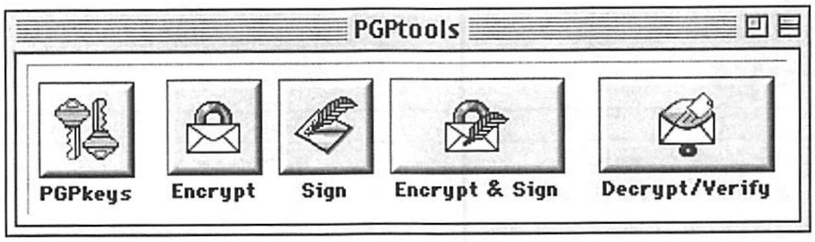


Figure 1

4.1 Visual metaphors

The metaphor of *keys* is built into cryptologic terminology, and PGP's user interface relies heavily on graphical depictions of keys and locks. The PGPTools display, shown in Figure 1, offers four buttons to the user, representing four operations: Encrypt, Sign, Encrypt & Sign, and Decrypt/Verify, plus a fifth button for invoking the PGPKeys application. The graphical labels on these buttons indicate the encryption operation with an icon of a sealed envelope that has a metal loop on top to make it look like a closed padlock, and, for the decryption operation, an icon of an open envelope with a key inserted at the bottom. Even for a novice user, these appear to be straightforward visual metaphors that help make the use of keys to encrypt and decrypt into an intuitive concept.

Still more helpful, however, would be an extension of the metaphor to distinguish between public keys for encryption and private keys for decryption; normal locks use the same key to lock and unlock, and the key metaphor will lead people to expect the same for encryption and decryption if it is not visually clarified in some way. Faulty intuition in this case may lead them to assume that they can always decrypt anything they have encrypted, an assumption which may have upsetting consequences. Different icons for public and private keys, perhaps drawn to indicate that they fit together like puzzle pieces, might be an improvement.

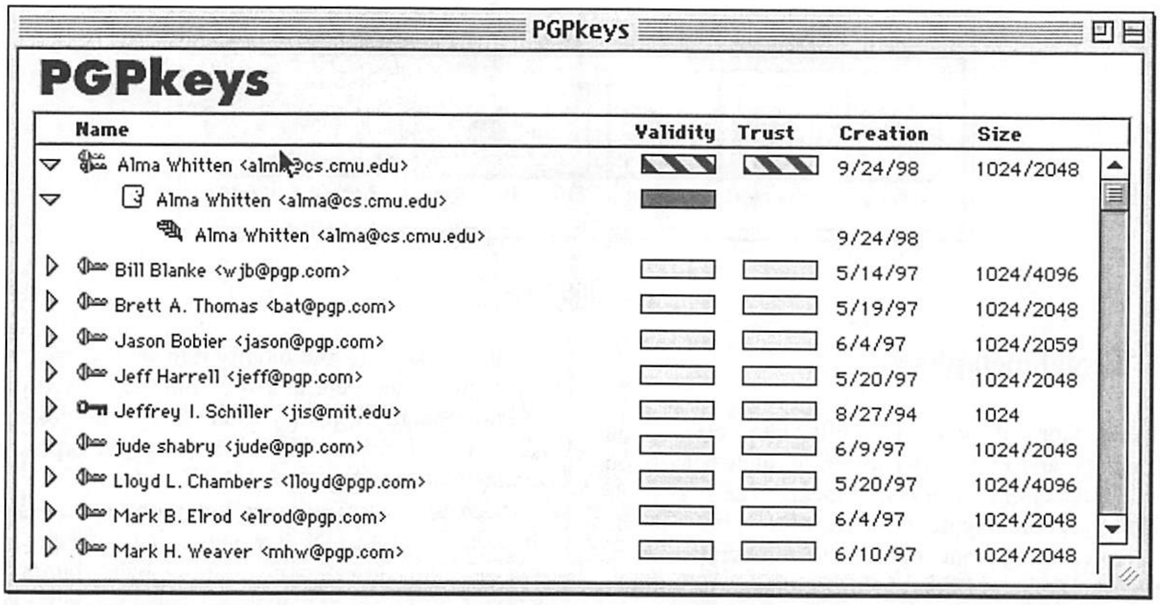
Signatures are another metaphor built into cryptologic terminology, but the icon of the blue quill pen that is used to indicate signing is problematic. People who are not familiar with cryptography probably know that quills are used for signing, and will recognize that the picture indicates the signature operation, but what they also need to understand is that they are using their private keys to generate signatures. The quill pen icon, which has nothing key-like about it, will not help them understand this and may even lead them to think that, along with the key objects that they use to encrypt, they also have quill pen objects that they use to sign. Quill pen icons encountered elsewhere in the program may be taken to be those objects, rather than the signatures that they are actually intended to represent. A better icon design might keep the quill pen to represent signing, but modify it to show a private key as the nib of the pen, and use some entirely different icon for signatures, perhaps something that looks more like a bit of inked handwriting and incorporates a keyhole shape.

Signature verification is not represented visually, which is a shame since it would be easy for people to overlook it altogether. The single button for Decrypt/Verify, labeled with an icon that only evokes decryption, could easily lead people to think that "verify" just means "verify that the decryption occurred correctly." Perhaps an icon that showed a private key unlocking the envelope and a public key unlocking the signature inside could suggest a much more accurate model to the user, while still remaining simple enough to serve as a button label.

4.2 Different key types

Originally, PGP used the popular RSA algorithm for encryption and signing. PGP 5.0 uses the Diffie-Hellman/DSS algorithms. The RSA and Diffie-Hellman/DSS algorithms use correspondingly different types of keys. The makers of PGP would prefer to see all the users of their software switch to use of Diffie-Hellman/DSS, but have designed PGP 5.0 to be backward compatible and handle existing RSA keys when necessary. The lack of forward compatibility, however, can be a problem: if a file is encrypted for several recipients, some of whom have RSA keys and some of whom have Diffie-Hellman/DSS keys, the recipients who have RSA keys will not be able to decrypt it unless they have upgraded to PGP 5.0; similarly, those recipients will not be able to verify signatures created with Diffie-Hellman/DSS without a software upgrade.

PGP 5.0 alerts its users to this compatibility issue in two ways. First, it uses different icons to depict the different key types: a blue key with an old fashioned shape for RSA keys, and a brass key with a more modern shape for Diffie-Hellman/DSS keys, as shown in Figure 2. Second, when users attempt to encrypt documents using mixed key types, a warning message



PGPkeys

Name	Validity	Trust	Creation	Size
Alma Whitten <alma@cs.cmu.edu>	▨	▨	9/24/98	1024/2048
Alma Whitten <alma@cs.cmu.edu>	■			
Alma Whitten <alma@cs.cmu.edu>			9/24/98	
Bill Blanke <wjb@pgp.com>	□	□	5/14/97	1024/4096
Brett A. Thomas <bat@pgp.com>	□	□	5/19/97	1024/2048
Jason Bobier <jason@pgp.com>	□	□	6/4/97	1024/2059
Jeff Harrell <jeff@pgp.com>	□	□	5/20/97	1024/2048
Jeffrey I. Schiller <jis@mit.edu>	□	□	8/27/94	1024
jude shabry <jude@pgp.com>	□	□	6/9/97	1024/2048
Lloyd L. Chambers <lloyd@pgp.com>	□	□	5/20/97	1024/4096
Mark B. Elrod <elrod@pgp.com>	□	□	6/4/97	1024/2048
Mark H. Weaver <mhw@pgp.com>	□	□	6/10/97	1024/2048

Figure 2

is displayed to tell them that recipients who have earlier versions of PGP may not be able to decrypt it.

Unfortunately, information about the meaning of the blue and brass key icons is difficult to find, requiring users either to go looking through the 132 page manual, or to figure it out based on the presence of other key type data. Furthermore, other than the warning message encountered during encryption, explanation of why the different key types are significant (in particular, the risk of forward compatibility problems) is given only in the manual. Double-clicking on a key pops up a Key Properties window, which would be a good place to provide a short message about the meaning of the blue or brass key icon and the significance of the corresponding key type.

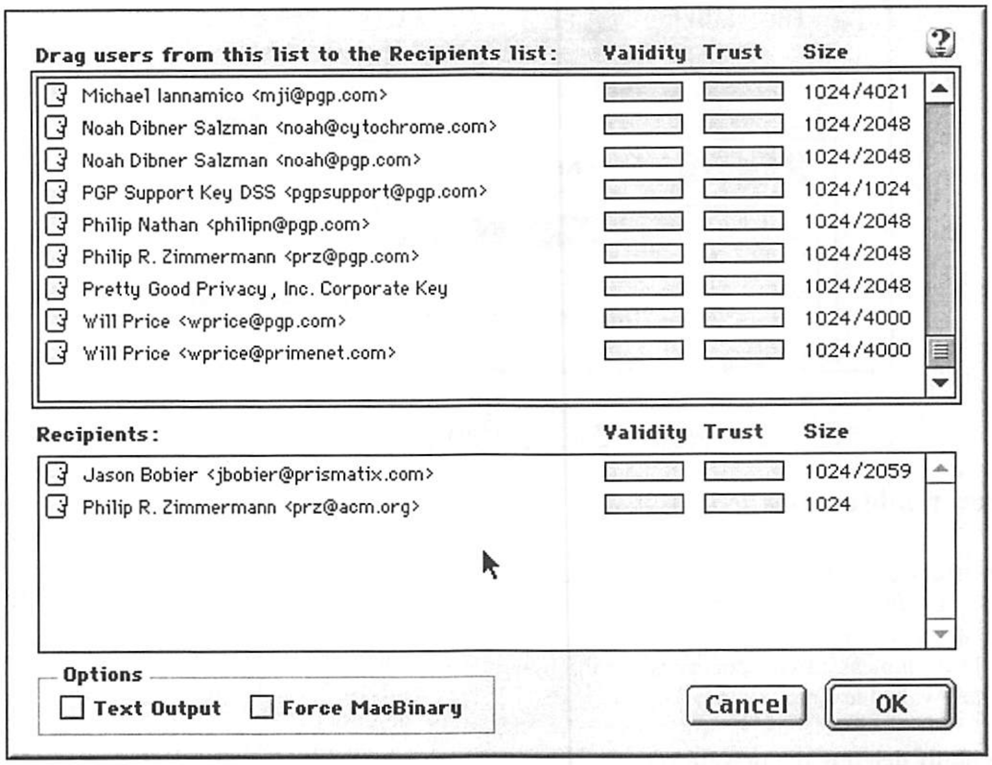
It is most important for the user to pay attention to the key types when choosing a key for message encryption, since that is when mixed key types can cause compatibility problems. However, PGP's dialog box (see Figure 3) presents the user with the metaphor of choosing people (recipients) to receive the message, rather than keys to encrypt the message with. This is not a good design choice, not only because the human head icons obscure the key type information, but also because people may have multiple keys, and it is counterintuitive for the dialog to display multiple versions of a person rather than the multiple keys that person owns.

4.3 Key server

Key servers are publicly accessible (via the Internet) databases in which anyone can publish a public key joined to a name. PGP is set to access a key server at MIT by default, but there are others available, most of which are kept up to date as mirrors of each other. PGP offers three key server operations to the user under the Keys pull-down menu shown in Figure 4: Get Selected Key, Send Selected Key, and Find New Keys. The first two of those simply connect to the key server and perform the operation. The third asks the user to type in a name or email address to search for, connects to the key server and performs the search, and then tells the user how many keys were returned as a result, asking whether or not to add them to the user's key ring.

The first problem we find with this presentation of the key server is that users may not realize it exists, since there is no representation of it in the top level of the PGPKeys display. Putting the key server operations under a Key Server pull-down menu would be a better design choice, especially since it is worthwhile to encourage the user to make a mental distinction between operations that access remote machines and those that are purely local. We also think that it should be made clearer that a remote machine is being accessed, and that the identity of the remote machine should be displayed. Often the "connecting...receiving data...closing connection" series of status messages that PGP displayed flashed by almost too quickly to be read.

At present, PGPKeys keeps no records of key server accesses. There is nothing to show whether a



Drag users from this list to the Recipients list:

	Validity	Trust	Size
Michael Iannamico <mji@pgp.com>	▨	▨	1024/4021
Noah Dibner Salzman <noah@cytochrome.com>	▨	▨	1024/2048
Noah Dibner Salzman <noah@pgp.com>	▨	▨	1024/2048
PGP Support Key DSS <pgpsupport@pgp.com>	▨	▨	1024/1024
Philip Nathan <philipn@pgp.com>	▨	▨	1024/2048
Philip R. Zimmermann <prz@pgp.com>	▨	▨	1024/2048
Pretty Good Privacy, Inc. Corporate Key	▨	▨	1024/2048
Will Price <wprice@pgp.com>	▨	▨	1024/4000
Will Price <wprice@primenet.com>	▨	▨	1024/4000

Recipients:

	Validity	Trust	Size
Jason Bobier <jbobier@prismatix.com>	▨	▨	1024/2059
Philip R. Zimmermann <prz@acm.org>	▨	▨	1024

Options: ☐ Text Output ☐ Force MacBinary

Figure 3

key has been sent to a key server, or when a key was fetched or last updated, and from which key server the key was fetched or updated. This is information that might be useful to the user for key management and for verifying that key server operations were completed successfully. Adding this record keeping to the information displayed in the Key Properties window would improve PGP.

Key revocation, in which a certificate is published to announce that a previously published public key should no longer be considered valid, generally implies the use of the key server to publicize the revocation. PGP's key revocation operation does not send the resulting revocation certificate to the key server, which is probably as it should be, but there is a risk that some users will assume that it does do so, and fail to take that action themselves. A warning that the created revocation certificate has not yet been publicized would be appropriate.

4.4 Key management policy

PGP maintains two ratings for each public key in a PGP key ring. These ratings may be assigned by the user or derived automatically. The first of these ratings is *validity* which is meant to indicate how sure the user is that the key is safe to encrypt with (i.e., that it does belong to the person whose name it is labeled with). A key may be labeled as completely valid, marginally valid, or invalid. Keys that the user generates are always completely valid. The second of these ratings is *trust* which indicates how much faith the user has in the key (and implicitly, the owner of the key) as a certifier of other keys. Similarly, a key may be labeled as completely trusted, marginally trusted, or untrusted, and the user's own keys are always completely trusted.

What the user may not realize, unless they read the manual very carefully, is that there is a policy built into PGP that automatically sets the validity rating of a key based on whether it has been signed by a certain number of sufficiently trusted keys. This is dangerous. There is nothing to prevent users from innocently assigning their own interpretations to those ratings and setting them accordingly (especially since "validity" and "trust" have different colloquial meanings), and it is certainly possible that some people might make mental use of the validity rating while disregarding and perhaps incautiously modifying the trust ratings. PGP's ability to automatically derive validity ratings can be useful, but the fact that PGP is doing so needs to be made obvious to the user.

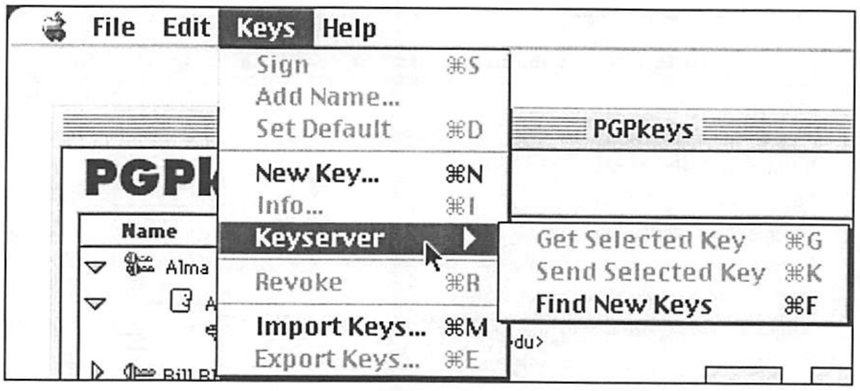


Figure 4

4.5 Irreversible actions

Some user errors are reversible, even if they require some time and effort to reconstruct the desired state. The ones we list below, however, are not, and potentially have unpleasant consequences for the user, who might lose valuable data.

Accidentally deleting the private key

A public key, if deleted, can usually be gotten again from a key server or from its owner. A private key, if deleted and not backed up somewhere, is gone for good, and anything encrypted with its corresponding public key will never be able to be decrypted, nor will the user ever be able to make a revocation certificate for that public key. PGP responds to any attempt to delete a key with the question "Do you really want to delete these items?" This is fine for a public key, but attempts to delete a private key should be met with a warning about the possible consequences.

Accidentally publicizing a key

Information can only be added to a key server, not removed. A user who is experimenting with PGP may end up generating a number of key pairs that are permanently added to the key server, without realizing that these are permanent entries. It is true that the effect of this can be partially addressed by revoking the keys later (or waiting for them to expire), but this is not a satisfactory solution. First, even if a key is revoked or expired, it remains on the key server. Second, the notions of revocation and expiration are relatively sophisticated concepts; concepts that are likely to be unfamiliar to a novice user. For example, as discussed above, the user may accidentally lose the ability to generate a revocation certificate for a key. This is particularly likely for a user who was experimenting with PGP and generating a variety of test keys that they intend to delete. One way to address this problem would be to warn the user when he sends a key to a server that the information being sent will be a permanent addition.

Accidentally revoking a key

Once the user revokes a public key, the only way to undo the revocation is to restore the key ring from a backup copy. PGP's warning message for the revocation operation asks "Are you sure you want to revoke this key? Once distributed, others will be unable to encrypt data to this key." This message doesn't warn the user that, even if no distribution has taken place, a previous backup of the key ring will be needed if the user wants to undo the revocation. Also, it may contribute to the misconception that revoking the key automatically distributes the revocation.

Forgetting the pass phrase

PGP suggests that the user make a backup revocation certificate, so that if the pass phrase is lost, at least the user can still use that certificate to revoke the public key. We agree that this is a useful thing to do, but we also believe that only expert users of PGP will understand what this means and how to go about doing so (under PGP's current design, this requires the user to create a backup of the key ring, revoke the public key, create another backup of the key ring that has the revoked key, and then restore the key ring from the original backup).

Failing to back up the key rings

We see two problems with the way the mechanism for backing up the key rings is presented. First, the user is not reminded to back up the key rings until he or she exits PGPKeys; it would be better to remind as soon as keys are generated, so as not to risk losing them to a system crash. Second, although the reminder message tells the user that it is important to back up the keys to some medium other than the main hard drive, the dialog box for backing up presents the main PGP folder as a default backup location. Since most users will just click the "Okay" button and accept the default, this is not a good design.

4.6 Consistency

When PGP is in the process of encrypting or signing a file, it presents the user with a status message that says it is currently "encoding." It would be better to say "encrypting" or "signing", since seeing terms that explicitly match the operations being performed helps to create a clear mental model for the user, and introducing a third term may confuse the user into thinking there is a third operation taking place. We recognize that the use of the term "encoding" here may simply be a programming error and not a design choice per se, but we think this is something that should be caught by usability-oriented product testing.

4.7 Too much information

In previous implementations of PGP, the supporting functions for key management (creating key rings, collecting other people's keys, constructing a "web of trust") tended to overshadow PGP's simpler primary functions, signing and encryption. PGP 5.0 separates these functions into two applications: PGPKeys for key management, and PGPTools for signing and encryption. This cleans up what was previously a rather jumbled collection of primary and supporting functions, and gives the user a nice simple interface to the primary functions. We believe, however, that the PGPKeys application still presents the user with far too much information to make sense of, and that it needs to do a better job of distinguishing between basic, intermediate, and advanced levels of key management activity so as not to overwhelm its users.

Currently, the PGPKeys display (see Figure 2) always shows the following information for each key on the user's key ring: owner's name, validity, trust level, creation date, and size. The key type is also indicated by the choice of icon, and the user can toggle the display of the signatures on each key. This is a lot of information, and there is nothing to help the user figure out which parts of the display are the most important to pay attention to. We think that this will cause users to fail to recognize data that is immediately relevant, such as the key type; that it will increase the chances that they will assign wrong interpretations to some of the data, such as trust and validity; and that it will add to making users feel overwhelmed and uncertain that they are managing their security successfully.

We believe that, realistically, the vast majority of PGP's users will be moving from sending all of their email in plain text to using simple encryption when they email something sensitive, and that they will be inclined to trust all the keys they acquire, because they are looking for protection against eavesdroppers and not against the sort of attack that would try to trick them into using false keys. A better design of PGPKeys would have an initial display configuration that concentrated on giving the user the correct model of the relationship between public and private keys, the significance of key types, and a clear understanding of the functions for acquiring and distributing keys. Removing the validity, trust level, creation date and size from the display would free up screen area for this, and would help the user focus on understanding the basic model well. Some security experts may find the downplaying of this information alarming, but the goal here is to enable users who are inexperienced with cryptography to understand and begin to use the basics, and to prevent confusion or frustration that might lead them to use PGP incorrectly or not at all.

A smaller set of more experienced users will probably care more about the trustworthiness of their keys; perhaps these users do have reason to believe that the contents of their email is valuable enough to be the target of a more sophisticated, planned attack, or perhaps they really do need to authenticate a digital signature as coming from a known real world entity. These users will need the information given by the signatures on each key. They may find the validity and trust labels useful for recording their assessments of those signatures, or they may prefer to glance at the actual signatures each time. It would be worthwhile to allow users to add the validity and trust labels to the display if they want to, and to provide easily accessible help for users who are transitioning to this more sophisticated level of use. But this would only make sense if the automatic derivation of validity by PGP's built-in policy were turned off for these users, for the reasons discussed in Section 4.4.

Key size is really only relevant to those who actually fear a cryptographic attack, and could certainly be left as information for the Key Properties dialog, as could the creation date. Users who are sophisticated enough to make intelligent use of that information are certainly sophisticated enough to go looking for it.

5 User test

5.1 Purpose

Our user test was designed to evaluate whether PGP 5.0 meets the specific usability standard described in Section 2.3. We gave our participants a test scenario that was both plausible and appropriately motivating, and then avoided interfering with their attempts to carry out the security tasks that we gave them.

5.2 Description

5.2.1 Test design

Our test scenario was that the participant had volunteered to help with a political campaign and had been given the job of campaign coordinator (the party affiliation and campaign issues were left to the participant's imagination, so as not to offend anyone). The participant's task was to send out campaign plan updates to the other members of the campaign team by email, using PGP for privacy and authentication. Since presumably volunteering for a political campaign implies a personal investment in the campaign's success, we hoped that the participants would be appropriately motivated to protect the secrecy of their messages.

Since PGP does not handle email itself, it was necessary to provide the participants with an email handling program to use. We chose to give them Eudora, since that would allow us to also evaluate the success of the Eudora plug-in that is included with PGP. Since we were not interested in testing the usability of Eudora (aside from the PGP plug-in), we gave the participants a brief Eudora tutorial before starting the test, and intervened with assistance during the test if a participant got stuck on something that had nothing to do with PGP.

After briefing the participants on the test scenario and tutoring them on the use of Eudora, they were given an initial task description which provided them with a secret message (a proposed itinerary for the candidate), the names and email addresses of the campaign manager and four other campaign team members, and a request to please send the secret message to the five team members in a signed and encrypted email. In order to complete this task, a participant had to generate a key pair, get the team members' public keys, make their own public key available to the team members, type the (short) secret message into an email, sign the email using their private key, encrypt the email using the five team members' public keys, and send the result. In addition, we designed the test so that one of the team members had an RSA key while the others all had Diffie-Hellman/DSS keys, so that if a participant encrypted one copy of the message for all five team members (which was the expected interpretation of the task), they would encounter the mixed key types warning message. Participants were told that after accomplishing that initial task, they should wait to receive email from the campaign team members and follow any instructions they gave.

Each of the five campaign team members was represented by a dummy email account and a key pair which were accessible to the test monitor through a networked laptop. The campaign manager's private key was used to sign each of the team members' public keys, including her own, and all five of the signed public keys were placed on the default key server at MIT, so that they could be retrieved by participant requests.

Under certain circumstances, the test monitor posed as a member of the campaign team and sent email to the participant from the appropriate dummy account. These circumstances were:

1. The participant sent email to that team member asking a question about how to do something. In that case, the test monitor sent the minimally informative reply consistent with the test scenario, i.e. the minimal answer that wouldn't make that team member seem hostile or ignorant beyond the bounds of plausibility[3].

2. The participant sent the secret in a plaintext email. The test monitor then sent email posing as the campaign manager, telling the participant what

[3] This aspect of the test may trouble the reader in that different test participants were able to extract different amounts of information by asking questions in email, thus leading to test results that are not as standardized as we might like. However, this is in some sense realistic; PGP is being tested here as a utility for secure communication, and people who use it for that purpose will be likely to ask each other for help with the software as part of that communication. We point out also that the purpose of our test is to locate extreme usability problems, not to compare the performance of one set of participants against another, and that while inaccurately improved performance by a few participants might cause us to fail to identify some usability problems, it certainly would not lead us to identify a problem where none exists.

happened, stressing the importance of using encryption to protect the secrets, and asking the participant to try sending an encrypted test email before going any further. If the participant succeeded in doing so, the test monitor (posing as the campaign manager) then sent an updated secret to the participant in encrypted email and the test proceeded as from the beginning.

3. The participant sent email encrypted with the wrong key. The test monitor then sent email posing as one of the team members who had received the email, telling the participant that the team member was unable to decrypt the email and asking whether the participant had used that team member's key to encrypt.

4. The participant sent email to a team member asking for that team member's key. The test monitor then posed as that team member and sent the requested key in email.

5. The participant succeeded in carrying out the initial task. They were then sent a signed, encrypted email from the test monitor, posing as the campaign manager, with a change for the secret message, in order to test whether they could decrypt and read it successfully. If at that point they had not done so on their own, they received email prompting to remember to back up their key rings and to make a backup revocation certificate, to see if they were able to perform those tasks. If they had not sent a separately encrypted version of the message to the team member with the RSA key, they also received email from the test monitor posing as that team member and complaining that he couldn't decrypt the email message.

6. The participant sent email telling the team member with the RSA key that he should generate a new key or should upgrade his copy of PGP. In that case the test monitor continued sending email as that team member, saying that he couldn't or didn't want to do those things and asking the participant to please try to find a way to encrypt a copy that he could decrypt.

Each test session lasted for 90 minutes, from the point at which the participant was given the initial task description to the point when the test monitor stopped the session. Manuals for both PGP and Eudora were provided, along with a formatted floppy disk, and participants were told to use them as much as they liked.

5.2.2 Participants

The user test was run with twelve different participants, all of whom were experienced users of email, and none of whom could describe the difference between public and private key cryptography prior to the test sessions. The participants all had attended at least some college, and some had graduate degrees. Their ages ranged from 20 to 49, and their professions were diversely distributed, including graphic artists, programmers, a medical student, administrators and a writer. More detailed information about participant selection and demographics is available in [18].

5.3 Results

We summarize the most significant results we observed from the test sessions, again focusing on the usability standard for PGP that we gave in Section 2.3. Detailed transcripts of the test sessions are available in [18].

Avoiding dangerous errors

Three of the twelve test participants (P4, P9, and P11) accidentally emailed the secret to the team members without encryption. Two of the three (P9 and P11) realized immediately that they had done so, but P4 appeared to believe that the security was supposed to be transparent to him and that the encryption had taken place. In all three cases the error occurred while the participants were trying to figure out the system by exploring.

One participant (P12) forgot her pass phrase during the course of the test session and had to generate a new key pair. Participants tended to choose pass phrases that could have been standard passwords, eight to ten characters long and without spaces.

Figuring out how to encrypt with any key

One of the twelve participants (P4) was unable to figure out how to encrypt at all. He kept attempting to find a way to "turn on" encryption, and at one point believed that he had done so by modifying the settings in the Preferences dialog in PGPKeys. Another of the twelve (P2) took more than 30 minutes[4] to figure out how to encrypt, and the method he finally found required a reconfiguration of PGP (to make it display the PGPMenu inside Eudora). Another (P3) spent 25

[4] This is measured as time the participant spent working on the specific task of encrypting a message, and does not include time spent working on getting keys, generating keys, or otherwise exploring PGP and Eudora.

minutes sending repeated test messages to the team members to see if she had succeeded in encrypting them (without success), and finally succeeded only after being prompted to use the PGP Plug-In buttons.

Figuring out the correct key to encrypt with

Among the eleven participants who figured out how to encrypt, failure to understand the public key model was widespread. Seven participants (P1, P2, P7, P8, P9, P10 and P11) used only their own public keys to encrypt email to the team members. Of those seven, only P8 and P10 eventually succeeded in sending correctly encrypted email to the team members before the end of the 90 minute test session (P9 figured out that she needed to use the campaign manager's public key, but then sent email to the the entire team encrypted only with that key), and they did so only after they had received fairly explicit email prompting from the test monitor posing as the team members. P1, P7 and P11 appeared to develop an understanding that they needed the team members' public keys (for P1 and P11, this was also after they had received prompting email), but still did not succeed at correctly encrypting email. P2 never appeared to understand what was wrong, even after twice receiving feedback that the team members could not decrypt his email.

Another of the eleven (P5) so completely misunderstood the model that he generated key pairs for each team member rather than for himself, and then attempted to send the secret in an email encrypted with the five public keys he had generated. Even after receiving feedback that the team members were unable to decrypt his email, he did not manage to recover from this error.

Decrypting an email message

Five participants (P6, P8, P9, P10 and P12) received encrypted email from a team member (after successfully sending encrypted email and publicizing their public keys). P10 tried for 25 minutes but was unable to figure out how to decrypt the email. P9 mistook the encrypted message block for a key, and emailed the team member who sent it to ask if that was the case; after the test monitor sent a reply from the team member saying that no key had been sent and that the block was just the message, she was then able to decrypt it successfully. P6 had some initial difficulty viewing the results after decryption, but recovered successfully within 10 minutes. P8 and P12 were able to decrypt without any problems.

Publishing the public key

Ten of the twelve participants were able to successfully make their public keys available to the team members; the other two (P4 and P5) had so much difficulty with earlier tasks that they never addressed key distribution. Of those ten, five (P1, P2, P3, P6 and P7) sent their keys to the key server, three (P8, P9 and P10) emailed their keys to the team members, and P11 and P12 did both. P3, P9 and P10 publicized their keys only after being prompted to do so by email from the test monitor posing as the campaign manager.

The primary difficulty that participants appeared to experience when attempting to publish their keys involved the iconic representation of their key pairs in PGPKeys. P1, P11 and P12 all expressed confusion about which icons represented their public keys and which their private keys, and were disturbed by the fact that they could only select the key pair icon as an indivisible unit; they feared that if they then sent their selection to the key server, they would be accidentally publishing their private keys. Also, P7 tried and failed to email her public key to the team members; she was confused by the directive to "paste her key into the desired area" of the message, thinking that it referred to some area specifically demarcated for that purpose that she was unable to find.

Getting other people's public keys

Eight of the twelve participants (P1, P3, P6, P8, P9, P10, P11 and P12) successfully got the team members' public keys; all of the eight used the key server to do so. Five of the eight (P3, P8, P9, P10 and P11) received some degree of email prompting before they did so. Of the four who did not succeed, P2 and P4 never seemed aware that they needed to get the team members' keys, P5 was so confused about the model that he generated keys for the team members instead, and P7 spent 15 minutes trying to figure out how to get the keys but ultimately failed.

P7 gave up on using the key server after one failed attempt in which she tried to retrieve the campaign manager's public key but got nothing back (perhaps due to mis-typing the name). P1 spent 25 minutes trying and failing to import a key from an email message; he copied the key to the clipboard but then kept trying to decrypt it rather than import it. P12 also had difficulty trying to import a key from an email message: the key was one she already had in her key ring, and when her copy and paste of the key failed to have any effect on the PGPKeys display, she assumed that her attempt had failed and kept trying. Eventually she became so confused that she began trying to decrypt the key instead.

Handling the mixed key types problem

Four participants (P6, P8, P10 and P12) eventually managed to send correctly encrypted email to the team members (P3 sent a correctly encrypted email to the campaign manager, but not to the whole team). P6 sent an individually encrypted message to each team member to begin with, so the mixed key types problem did not arise for him. The other three received a reply email from the test monitor posing as the team member with an RSA key, complaining that he was unable to decrypt their email.

P8 successfully employed the solution of sending that team member an email encrypted only with his own key. P10 explained the cause of the problem correctly in an email to that team member, but didn't manage to offer a solution. P12 half understood, initially believing that the problem was due to the fact that her own key pair was Diffie-Hellman/DSS, and attempting to generate herself an RSA key pair as a solution. When she found herself unable to do that, she then decided that maybe the problem was just that she had a corrupt copy of that team member's public key, and began trying in various ways to get a good copy of it. She was still trying to do so at the end of the test session.

Signing an email message

All the participants who were able to send an encrypted email message were also able to sign the message (although in the case of P5, he signed using key pairs that he had generated for other people). It was unclear whether they assigned much significance to doing so, beyond the fact that it had been requested as part of the task description.

Verifying a signature on an email message

Again, all the participants who were able to decrypt an email message were by default also verifying the signature on the message, since the only decryption operation available to them includes verification. Whether they were aware that they were doing so, or paid any attention to the verification result message, is not something we were able to determine from this test.

Creating a backup revocation certificate

We would have liked to know whether the participants were aware of the good reasons to make a backup revocation certificate and were able to figure out how to do so successfully. Regrettably, this was very difficult to test for. We settled for direct prompting to make a backup revocation certificate, for participants who managed to successfully send encrypted email and decrypt a reply (P6, P8 and P12).

In response to this prompting, P6 generated a test key pair and then revoked it, without sending either the key pair or its revocation to the key server. He appeared to think he had successfully completed the task. P8 backed up her key rings, revoked her key, then sent email to the campaign manager saying she didn't know what to do next. P12 ignored the prompt, focusing on another task.

Deciding whether to trust keys from the key server

Of the eight participants who got the team members' public keys, only three (P1, P6, and P11) expressed some concern over whether they should trust the keys. P1's worry was expressed in the last five minutes of his test session, so he never got beyond that point. P6 noted aloud that the team members' keys were all signed by the campaign manager's key, and took that as evidence that they could be trusted. P11 expressed great distress over not knowing whether or not she should trust the keys, and got no further in the remaining ten minutes of her test session. None of the three made use of the validity and trust labeling provided by PGPKeys.

6 Conclusions

6.1 Failure of standard interface design

The results seen in our case study support our hypothesis that the standard model of user interface design, represented here by PGP 5.0, is not sufficient to make computer security usable for people who are not already knowledgeable in that area. Our twelve test participants were generally educated and experienced at using email, yet only one third of them were able to use PGP 5.0 to correctly sign and encrypt an email message when given 90 minutes in which to do so. Furthermore, one quarter of them accidentally exposed the secret they were meant to protect in the process, by sending it in email they thought they had encrypted but had not.

In Section 2.1, we defined usability for security in terms of four necessary qualities, which translate directly to design priorities. PGP 5.0's user interface fails to enable effective security where it is not designed in accordance with those priorities: test participants did not understand the public key model well enough to know that they must get public keys for people they wish to send secure email to; many who knew that they needed to get a key or to encrypt still had substantial difficulties in figuring out how to do so; some erroneously sent secrets in plaintext, thinking that they had encrypted; and many expressed frustration and unhappiness with the experience of trying to use PGP 5.0, to the point where it is unlikely that they would have continued to use it in the real world.

All this failure is despite the fact that PGP 5.0 is attractive, with basic operations neatly represented by buttons with labels and icons, and pull-down menus for the rest, and despite the fact that it is simple to use for those who already understand the basic models of public key cryptography and digital signature-based trust. Designing security that is usable enough to be effective for those who don't already understand it must thus require something more.

6.2 Usability evaluation for security

Since usable security requires user interface design priorities that are not the same as those of general consumer software, it likewise requires usability evaluation methods that are appropriate to testing whether those priorities have been sufficiently achieved. Standard usability evaluation methods, simplistically applied, may treat security functions as if they were primary rather than secondary goals for the user, leading to faulty conclusions. A body of public work on usability evaluation in a security context would be extremely valuable, and will almost certainly have to come from research sources, since software developers are not eager to make public the usability flaws they find in their own products.

In our own work, which has focused on personal computer users who have little initial understanding of security, we have assigned a high value to learnability, and thus have found cognitive walkthrough to be a natural evaluation technique. Other techniques may be more appropriate for corporate or military users, but are likely to need similar adaptation to the priorities appropriate for security. In designing appropriate user tests, it may be valuable to look to other fields in which there is an established liability for consumer safety; such fields are more likely to have a body of research on how best to establish whether product designs successfully promote safe modes of use.

6.3 Toward better design strategies

The detailed findings in our case study suggest several design strategies for more usable security, which we are pursuing in our ongoing work. To begin with, it is clear that there is a need to communicate an accurate conceptual model of the security to the user as quickly as possible. The smaller and simpler that conceptual model is, the more plausible it will be that we can succeed in doing so. We thus are investigating pragmatic ways of paring down security functionality to that which is truly necessary and appropriate to the needs of a given demographic, without sacrificing the integrity of the security offered to the user.

After a minimal yet valid conceptual model of the security has been established, it must be communicated to the user, more quickly and effectively than has been necessary for conceptual models of other types of software. We are investigating several strategies for accomplishing this, including the possibility of carefully crafting interface metaphors to match security functionality at a more demanding level of accuracy.

In addition, we are looking to current research in educational software for ideas on how best to guide users through learning to manage their security. We do not believe that home users can be made to cooperate with extensive tutorials, but we are investigating gentler methods for providing users with the right guidance at the right time, including how best to make use of warning messages, wizards, and other interactive tools.

7 Related work

We have found very little published research to date on the problem of usability for security. Of what does exist, the most prominent example is the Adage project [12, 20], which is described as a system designed to handle authorization policies for distributed applications and groups. Usability was a major design goal in Adage, but it is intended for use by professional system administrators who already possess a high level of expertise, and as such it does not address the problems posed in making security effectively usable by a more general population. Work has also been done on the related issue of usability for safety critical systems [10], like those which control aircraft or manufacturing plants, but we may hope that unlike the users of personal computer security, users of those systems will be carefully selected and trained.

Ross Anderson discusses the effects of user non-compliance on security in [1], and Don Davis analyzes the unrealistic expectations that public-key based security systems often place on users in [3].

Beyond that, we know only of one paper on usability testing of a database authentication routine [8], and some brief discussion of the security and privacy issues inherent in computer supported collaborative work [16]. John Howard's thesis [6] provides interesting analyses of the security incidents reported to CERT[5] between 1989 and 1995, but focuses more on

[5] CERT is the Computer Emergency Response Team formed by the Defense Advanced Research Projects Agency, and located at Carnegie Mellon University.

the types of attacks than on the causes of the vulnerabilities that those attacks exploited, and represents only incidents experienced by entities sophisticated enough to report them to CERT.

Acknowledgements

We thank Robert Kraut for helpful advice on the design of our user test. This publication was supported in part by Contract No. 102590-98-C-3513 from the United States Postal Service. The contents of this publication are solely the responsibility of the authors.

References

1. Ross Anderson. Why Cryptosystems Fail. In *Communications of the ACM*, 37(11), 1994.

2. Matt Bishop. *UNIX Security: Threats and Solutions*. Presentation to SHARE 86.0, March 1996. At `http://seclab.cs.ucdavis.edu/~bishop/scriv/1996-share86.pdf`.

3. Don Davis. Compliance Defects in Public-Key Cryptography. In *Proceedings of the 6th USENIX Security Symposium*, 1996.

4. The Economist. *The End of Privacy*. May 1, 1999, pages 21-23.

5. Simson Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly and Associates, 1995.

6. John D. Howard. *An Analysis of Security Incidents on the Internet 1989-1995*. Carnegie Mellon University Ph.D. thesis, 1997.

7. John, B. E., & Mashyna, M. M. (1997) Evaluating a Multimedia Authoring Tool with Cognitive Walkthrough and Think-Aloud User Studies. In *Journal of the American Society of Information Science*, 48 (9).

8. Clare-Marie Karat. Iterative Usability Testing of a Security Application. In *Proceedings of the Human Factors Society 33rd Annual Meeting*, 1989.

9. Stephen Kent. Security. In *More Than Screen Deep: Toward Every-Citizen Interfaces to the Nation's Information Infrastructure*. National Academy Press, Washington, D.C., 1997.

10. Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Company, 1995.

11. Jakob Nielsen. Heuristic Evaluation. In *Usability Inspection Methods*, John Wiley & Sons, Inc., 1994.

12. The Open Group Research Institute. *Adage System Overview*. Published on the web in July 1998 at `http://www.osf.org/www/adage/relatedwork.htm`

13. Pretty Good Privacy, Inc. *PGP 5.0 Features and Benefits*. Published in 1997 at `http://pgp.com/products/PGP50-fab.cgi`

14. Pretty Good Privacy, Inc. *User's Guide for PGP for Personal Privacy, Version 5.0 for the Mac OS*. Packaged with software, 1997.

15. Jeffrey Rubin. *Handbook of usability testing: how to plan, design, and conduct effective tests*. Wiley, 1994.

16. HongHai Shen and Prasun Dewan. Access Control for Collaborative Environments. In *Proceedings of CSCW '92*.

17. Cathleen Wharton, John Rieman, Clayton Lewis and Peter Polson. The Cognitive Walkthrough Method: A Practioner's Guide. In *Usability Inspection Methods*, John Wiley & Sons, Inc., 1994.

18. Alma Whitten and J.D. Tygar. *Usability of Security: A Case Study*. Carnegie Mellon University School of Computer Science Technical Report CMU-CS-98-155, December 1998. `ftp://reports-archive.adm.cs.cmu.edu/1998/CMU-CS-98-155.ps`

19. Wogalter, M. S., & Young, S. L. (1994). Enhancing warning compliance through alternative product label designs. *Applied Ergonomics*, 25, 53-57.

20. Mary Ellen Zurko and Richard T. Simon. *User-Centered Security*. New Security Paradigms Workshop, 1996.

Jonah: Experience Implementing PKIX Reference Freeware

Mary Ellen Zurko[1], John Wray[1], Ian Morrison[2], Mike Shanzer[1], Mike Crane[2], Pat Booth[3], Ellen McDermott[2], Warren Macek[1], Ann Graham[2], Jim Wade[2], and Tom Sandlin[2]

[1] *Iris Associates*
5 Technology Park Drive
Westford, MA 01886

[2] *IBM*
[3] *Lotus*
Contact email: mzurko@iris.com

Abstract

IBM made a business decision to promote an open, secure, interoperable and integrateable environment by embracing the IETF's Public Key Infrastructure (X.509) standards, developing a reference implementation of the core set, and giving it away as freeware. We discuss the architecture of this reference code, emphasizing lessons learned from implementing the standards in an integrated PKI system.

1 Introduction

The explosion of the Internet and the Web has forced the distribution of information and resources. Various types of trust relationships must span these resources and the applications that use them. Application and middleware developers have responded to the challenge by creating their own security solutions, which are unfortunately non-interoperable. Thus, organizations of every kind are finding the management of security to be difficult and costly and interoperability of their applications impossible without major intervention. From a business perspective, with projected growth of business to business Internet commerce going from $8 billion in 1997 to $327 billion by 2002, it was seen by IBM as crucial that the infrastructure be in place to provide an open, secure, interoperable, and integrateable environment. Unfortunately, the trend so far has been for vendors to go their own way, trying to follow standards as they emerge, and otherwise interoperate in a pairwise fashion as the market dictates.

Customer feedback and internal technical advice caused us to focus on creating a common Public Key Infrastructure (PKI). IBM decided that the best strategy to fulfill its business requirements was to find the most mature standard for PKI available and throw its support behind that standard, by developing and making available a freeware reference implementation, and integrating that standard into its products. A freely available reference implementation allows potential integrators to try it out without a large investment, and provide a baseline for interoperability testing. In addition, feedback from the development experience enhances the quality of the specifications. By providing leadership within the industry, IBM believes it advances Internet computing and ultimately drives faster acceptance/implementation of electronic business. IBM's goal is to help the industry converge on one security infrastructure for the web, thus allowing faster deployment of a secure interoperable underpinning for electronic business. We have learned over the years that disparate infrastructures slow evolution and acceptance of technology; the true market is in applications that exploit common infrastructures.

After surveying the body of work already done in this area, as well as the realities of the commercial world, we chose the Internet Engineering Task Force's (IETF) Public Key Infrastructure (PKIX [PKIX]) drafts as our architectural basis. These standards were the beneficiaries of much hard work on the part of many companies, and included the X.509v3 certificate format, which had emerged as a de facto standard. They cover the certificate life-cycle: enrollment and initial certification, key-pair update, certificate update, certificate and Certificate Revocation List (CRL) publication, and certificate revocation. Jonah, our freeware project, currently implements the following PKIX standards: Internet X.509 Public Key Infrastructure Certificate Management Protocols [RFC2510], Certificate Request Message Format [RFC2511], Internet Public Key Infrastructure X.509 Certificate and CRL Profile [RFC2459], and Internet X.509 Public Key Infrastructure LDAPv2 Schema [LDAP98]. Besides implementing PKIX standards, Jonah includes code for Common Data Security Architecture (CDSA) [CDSA97] from Intel, which has been accepted as a standard by The Open Group. CDSA is a framework that can support multiple security providers. The code is a freeware version of IBM's Keyworks product.

IBM then put together its first Iris/Lotus/IBM cross company team. The team consisted mostly of a group of engineers, representing different security and technical specialties, who had different ideas about PKI, yet were able to create a reference implementation using the IETF RFCs that were available at the time. This team takes advantage of Iris' Domino PKI experience and the pool of talent in PKIs in the Massachusetts area, Lotus' expertise in security product management, and IBM's long history of cryptography and security standards experience, as well as their broad pool of internal expertise. The next section discusses how the architecture was designed and implemented, in spite of moving specifications that had gaps. With Jonah one can run an EE, an RA and a CA. The architectural discussion covers our design for ASN.1 support, implementation of the transaction protocol and extensions to support RA to CA communication, architectural support for portability, use of CDSA as an abstraction layer to cryptography and data storage, our server API and persistent storage support. We then discuss trust policy, also accessed via CDSA, including our support for chain building (which is not specified in the PKIX standards). The discussion of smart card usage includes our layering of PKCS#11 and CDSA. The challenges of implementing a freeware Graphical User Interface (GUI) are outlined in the next section. After that, the section on porting issues discusses how we are porting our NT code to UNIX, and the section on testing emphasizes the issues that arose with testing freeware. We conclude with a look at current status and issues.

2 Description and Evolution of the Jonah Server Architecture

In this section we describe the Jonah architecture as it evolved over time. The architecture was affected by both high- and low-level considerations. The PKIX architecture [RFC2459] describes three PKI components that are involved during enrollment: the End Entity (EE), the Registration Authority (RA), and the Certification Authority (CA). The End Entity acts as the agent of the end-user, participating in the enrollment and other protocols with the RA on the user's behalf. The RA is responsible for enforcing most policy decisions (for example, choice of a distinguished name for the user). The CA is responsible for taking final actions that can occur offline (such as actually creating and signing a certificate).

Since certificates and all PKIX protocols are specified using ASN.1 [ASN.1], a component was needed to perform ASN.1 parsing and generation. A commercial ASN.1 compiler could not be used for freeware. Instead, we opted to construct a C++ class library to handle these tasks. The class library directly implements basic ASN.1 types (e.g., character string) and provides constructs for building up more complicated objects (e.g., sequences). The primary design goal of the class library was that the resulting C++ structures should be programmer-friendly. They should manage their own memory, provide straightforward access to the values within an ASN.1 stream, and hide from the programmer as much as possible ASN.1 concepts (like optional and default values) that relate more to the encoding of values than to their use. The class library enables the construction of high-level objects (like certificates) that know how to encode themselves as an ASN.1 Distinguished Encoding Rules (DER) stream and set their values from an ASN.1 Basic Encoding Rules (BER) stream [ISO8825]. This allows us to treat certificates and protocol messages as straightforward C++ objects, eliminating the need for a specialized "syntax" layer that converts between BER-encoded external messages and internal C++ objects. This decision had a significant impact on the design and implementation of the rest of Jonah. Having a single consistent internal representation of all the high-level PKI constructs as first-class objects greatly simplifies the software, and allows the developers to easily understand one another's code. The library's ease of use is demonstrated by the fact that we chose to use it for encoding much of our on-disk storage (see Section 2.1.2), in spite of there being no requirement to use ASN.1 encoding for purely local storage. In addition, groups within IBM are freely integrating the Jonah ASN.1 objects into their products.

2.1 Certificate Management Protocol Transaction Support

2.1.1 Certificate Management Protocols

PKIX defines protocols by which an EE can communicate with an RA. At the time that the Jonah project commenced, only Certificate Management Protocols (CMP) was at a stage suitable for implementation. CMP is derived from the Entrust PKI management protocol. It targets management functions for the entire certificate/key life. CMP defines message formats in the Certificate Request Message Format (CRMF), and describes various transports and how message protection should be done independent of transport. CMP is the only registration protocol that Jonah currently implements.

The other PKIX protocol is Certificate Management Protocol using CMS (CMC [CMC98]). It is co-authored by representatives of Netscape and Microsoft (among others). Its purpose is to be compatible with PKCS #7 [PKCS7] wrapped responses and PKCS #10 [PKCS10] certificate requests which web browsers currently use. The authors of Certificate Management Messages over CMS (CMC) have also added support for CRMF. CMC attempts to finish all transactions in a single round trip, an important consideration for the Internet where bandwidth and lagtime can be important. At time CMP has become an RFC, while CMC is still a internet-draft.

Also, CMP includes its own message protection, so it does not rely on a secure transport for message security. The CMP specification includes a description of TCP, SMTP, or HTTP as transport mechanisms. We chose to implement TCP as the primary transport for CMP, but want to allow additional transport mechanisms to be added reasonably easily. Thus we created an abstraction layer over sockets that could support additional transports. Jonah plans to implement the following CMP messages: Certificate Request/Reply, Revocation Request/Reply, CRL Announce, Proof of Possession Request/Reply, Cross Certification Request/Response, CA Key Update, Confirmation, and our extended General Messages. The EE only talks to the RA. Since the CA can be offline at any given time, or located on an isolated network, direct communications to the CA are kept to a minimum. (For the same reason, all LDAP interactions happen on the RA.)

After we decided to use CMP for EE communications, we needed to decide how the RA and the CA should share information. Nothing in PKIX specified such a protocol. We had the choice of specifying a new protocol or extending an existing one. Since we were implementing CMP and it seemed reasonably full featured and extensible, we decided to use CMP as the RA to CA protocol. In addition, since this is a standard published protocol it would be possible for others to write RAs and CAs that interoperated with ours. So far, we have only had to extend CMP in two areas, CRL requests and RA enrollment (RA to CA association). While CMP supports RA initiated certificate revocation, there is no message defined to allow an RA to request an on-demand CRL. Secondly, while CMP defines the RA and CA, it does not describe how they are introduced. For RA enrollment, we use the standard certificate request message, extending its control information to indicate that it is a request to become an RA for the target CA. We also defined a certificate extension that indicates the certificate is for one of the issuer's RAs. We expect this extension to be useful when an end-entity is looking for RAs. We added new general messages for an RA request a CA to generate and publish a new CRL, and for announcing that an RA is de-enrolling from a CA. We would like to standardize the extensions necessary for extending CMP to RA/CA communications.

2.1.2 Object-Store

Since the enrollment process may require human intervention at any of the three servers, a round-trip enrollment can take a significant amount of time. The system had to be designed to withstand any of its servers being stopped and restarted, without loss of transaction data. For this, each server has a disk-based *object-store* where short-term state data is stored. For example, an enrollment request is constructed at the End Entity by creating an empty request in its object-store, setting various fields in the request (e.g. SubjectName), then submitting the completed request to the RA. When the request is submitted, the object in the EE's object-store is marked as a *surrogate*, indicating that the real object is active on another server. Since CMP is a polling protocol when operating over TCP, the surrogate object stores the data needed to poll the RA for a response. When the enrollment succeeds, the resulting certificate is retrieved and the surrogate object becomes active once again.

Jonah views the object-store as a series of storage locations, each of which can hold an object. The objects stored in the object-store are constructed using the ASN.1 class library discussed earlier in Section 2. They are a sequence of a CMP message and additional control information such as sender and recipient names or addresses. To minimize the ASN.1 parsing overhead, Jonah uses a layer above the object-store that implements a write-through cache of object-store objects. Generally, this means that an object-store object need be parsed only the first time it is referenced by the Jonah code after a server restart. This object-cache layer also provides an additional per-object storage area that is not backed by a disk file. This is used for storing transient, security-related information (for example, the password under which a CMP preregistration record is protected [PKCS5]). An additional feature provided by the object-cache is locking of object-store objects, protecting against simultaneous access by multiple threads.

2.1.3 CMP Issues

One of the goals of doing a reference implementation is to verify the standards against an operational environment. CMP's most outstanding shortcoming is its lack of support for the current de facto standard of PKCS #10 and PKCS #7, as discussed in Section 2.1. Beyond that we have found several small problems with implementing CMP. We discovered places where CMP is vague. The most notable was discussion of the support for multiple certificate requests and revocation requests in a single message. The specification supports them, but handling them is underspecified. More information is needed on mapping multiple replies to the requests. In addition, a compliant service may not support multiples. However, its response to a request with multiple entries is undefined. CMP also mandates many out-of-band steps when initiating a relationship with a new entity (for instance, an EE's first certificate request). An RA administrator may be easy to identify and find in a corporate environment. The approach is unlikely to scale well to the Internet.

We have several issues around CMP's use of time. The CMP TCP protocol uses an absolute time as a polling time instead of a relative time (see Section 2.5 for a description of polling). This makes the assumption that all machine clocks are fairly well synchronized. The polling is represented by a 32-bit integer. CMP does not specify whether it is signed or unsigned, which is an interoperability issue. This integer stores when the next poll should happen based on the number of seconds from January 1, 1970 GMT. If it is signed, it will run out of space in the year 2038. If it is unsigned, it does not run out of space until 2106. If this field was a relative time, the longest amount of time you could have between polling events would be around 136 years, give or take a few months.

The specification should make transaction IDs and polling references more homogenous and consistent. A message transaction is labeled with a transaction ID, an octet string that must be included with every message. It is used to track the transaction from server to server. The polling reference in the protocol is a 32-bit integer value set by the recipient used for associating the recipient's delayed response with the initial request. It would be much easier to track a request from EE to RA to CA and back if these two fields shared the same value. For example, our test team would like to drive many transactions per minute, and track them easily with a value they control. In addition, we use transaction ID to associate a certificate request with the password for it specified via out of band means to the RA. CMP leaves open the exact value used for this purpose.

The PKIX LDAPv2 Schema segregates the list of revoked CAs into Authority Revocation Lists (ARLs), while other revoked certificates are listed in CRLs. Both CMP and the CRL profile are silent about ARLs. We use the Issuing Distribution Point extension to indicate whether a particular CRL contains only user certificates (CRL) or only CA certificates (ARL). We would like to see this method of defining an ARL explicitly called out as such, to minimize the chance of future interoperability problems.

Finally, as is often the case with policy-based decisions, there is some useful information not included in certificate requests. The most obvious missing feature is that there is no algorithm-independent way of determining the key length of the private key associated with a public key supplied in the certificate request.

2.2 Portability Concerns: Threads, Messages and Configuration

Although the primary development platform for Jonah is Windows NT, portability to various UNIX platforms is essential, both for Jonah to succeed as a reference implementation, and also for internal consumption within IBM. The next stage of architectural refinement after protocol definition was to abstract three areas that can be significant sources of portability problems: threading, message catalogs, and configuration. The *threading* primitives used by Jonah are based on Draft 10 of the POSIX 1003.1c threading standard [ISO9945]. The only significant differences are that the **lock** primitive for mutexes has been extended with an optional *timeout* parameter (which allows a thread that has been waiting on a mutex for a significant amount of time to perform another task before re-trying to acquire the mutex, as well as permitting deadlock recovery) and that once-blocks are not supported. The Jonah primitives are expected to be implemented over whatever native thread services are available on the target platform. Implementing them on Windows NT was fairly straightforward.

Jonah uses a *message catalog*, based on the XPG.4 [UNIX] standard, for all its status codes, both internal and exposed. It is intended that a port to a platform with a real XPG.4 system would use the native catalog, rather than the lightweight implementation provided as part of the freeware. One unexpected complication on Windows NT was that the Microsoft C Run Time Library implementation of printf() does not support the positional parameters required for handling internationalized messages. The Jonah freeware includes an implementation of printf() that supports this feature.

Configuration is handled in Jonah via a Windows 3.1-style initialization file, encapsulated in a C++ class. A productized version of Jonah would most likely replace this implementation with a platform-specific configuration data repository (e.g., the system registry on Windows NT) and GUI access. One of the major uses of this file is to configure policy information indicating how the CA handles certificate requests and revocations, and how any application uses the trust policy code to determine the validity of a certificate chain (Section 3 discusses the latter use).

The RA does all the policy verification of the EE for the CA. The CA trusts the RA for policy decisions like identity verification. The CA also checks all requests against its policy, and may involve an administrator in the final decision. Certificate policy in the initialization file indicates which optional fields the CA supports and which fields the RA must specify the value for. Currently, Jonah CAs can specify whether key usage is supported (and whether it's required), and whether a Policy is required. Policy configuration also indicates whether the RA (or the EE) may specify the certificate validity start time, whether Policy is marked critical in certificates from a CA (and is therefore used to constrain chains of certificates), and which signature algorithms the CA supports. Certificate revocation policy at the RA specifies whether an EE may request the revocation of any certificate, only one of their own certificates, or no certificates. CRL policy at the CA specifies how often CRLs are issued, how long each is valid, and which algorithm should be used to sign them (if the CA maintains more than one type of valid CRL signing key).

Other policy configuration information includes the server's name and the names of its RAs or CAs, and the URLs used to communicate with them. These URLs are of the form pkix://hostname:port. The initialization file is also used for LDAP configuration information at the RA and for text to object identifier (OID) translation. The latter use allows for extensions to the processing of Distinguished Names and algorithms. General configuration information specifies which signing and encryption algorithms a CA supports. The administrator must be careful to specify only algorithms that are supported in software. This is one of several examples of configuration information that motivates the need for a GUI in the future. A GUI could query the CDSA interface (see Section 2.3) to determine what algorithms are supported in software. Another issue with the use of the initialization file is the need to keep CA policy values in synch between CAs and their supporting RAs. Managing configuration information dynamically would allow CAs to notify their RAs of policy changes.

2.3 CDSA

Jonah uses Intel's Common Data Security Architecture (CDSA [CDSA97]) for access to cryptographic and data storage services. CDSA is a framework that supports dynamic loading of modular, pluggable low-level service providers. The Common Security Services Manager (CSSM) layer provides a consistent API to the underlying service provider modules, provides management services for loading and unloading providers, and determines their capabilities. By adopting CDSA, we were able to use existing code for virtual smart card and directory access, as well as allowing easy switching between our two supported cryptographic libraries: The Cylink Foundation Suite and RSADSI's BSAFE [BSAFE]. In Jonah, we decided not to use the Certificate Library (CL) component, since the ASN.1 class library is both more powerful and easier to use than the CDSA CL API. The Jonah reference implementation includes a stripped-down version of IBM's KeyWorks Toolkit, which is a product implementation of CDSA 1.2. The reference implementation removes KeyWorks' value-add features such as mutual authentication of framework and providers, and key recovery features.

Although the CDSA API is powerful, it is not particularly easy to program. Each routine takes parameters that specify the particular provider(s) to use. In Jonah, we preferred to be able to set providers at startup based on information in the configuration file. In addition, the objects that are passed across the CDSA API are CDSA-defined data structures with associated memory management semantics, whereas the natural structures within Jonah are ASN.1 objects. Another drawback to CDSA, which persists with CDSA 2.0, is that applications generally need to know the details of which providers they're dealing with, e.g., the supported interfaces and data structure conventions. Several groups within IBM have tackled this problem by creating an object-oriented layer on top of CDSA that protects the application from these details and presents a uniform interface to CDSA. Supporting a new provider requires only an update to the object-oriented shim, not the calling application. To address all these problems, we wrote a set of wrapper APIs that encapsulate the CDSA Data Storage Library (DL) and Cryptographic Service Provider (CSP) APIs. (Our CDSA Trust Policy support is discussed in Section 3.) They are relatively lightweight shims that provide a great deal of functionality by leveraging the Jonah ASN C++ classes and existing CSP and DL providers. The Jonah Krypto Library (JKL) provides useful domain-related APIs like *JKL_SignCertificate*, *JKL_VerifyCRL*, and *JKL_GenerateRandomData*. It deals with application domain objects like certificates, CRLs, and public keys in Jonah ASN C++ classes. The Jonah Directory Library (JDL) also takes ASN objects as parameters. It can manipulate all of the PKIX LDAP schema object classes and attributes through the CDSA DL interface. We have extended the LDAP schema to include communications information (URLs) for contacting a CA. This is used for RA enrollment.

Although the architecture provides a framework for pluggable service providers, generally an application needs to know specifically which add-in providers it is dealing with. For example, while the Cylink and BSAFE CSPs present the same interfaces through CSSM, the data formats for each differ. The BSAFE CSP returns keys in Distinguished Encoding Rules (DER) format; the Cylink CSP returns the same objects in flattened uint32 length/data buffers. The calling application must be aware of the data formats expected for common cryptographic data like public keys, private keys, signed data, hashes, and so on. Applications that use CDSA generally need to develop an interface layer that provides easy-to-use, domain specific interfaces that encapsulate the low-level details of CSSM. The result is an unfortunate loss of generality (one of the goals of CSSM) and less plug-and-play functionality. The Jonah CDSA-wrapper layers are the types of middle-tier interfaces needed on top of CDSA to make it useful to a wide range of users.

2.4 Unified Server Architecture and API

Figure 1 illustrates the three Jonah PKIX servers. The EE is a lightweight service that does not require much of the storage and authorization support that exists in the RA and CA. Future versions can integrate stripped down EE functionality into a variety of applications. The RA is the heavyweight process that approves and forwards transactions and interacts with LDAP. The CA is separate to enable offline processing to protect its key, and to enable separation of duty between the RA and CA administrators. This three server model is supported by PKIX transactions.

There is overlap in the infrastructural services used by each server, so Jonah is built as a single logical "back-end", which can be configured as EE, RA or CA at run-time. The only difference between the three servers is the GUI that starts them (see Section 5). The GUI calls the API to configure the server as either an EE, an RA or a CA. This call is an obvious place to perform initialization functions in a layered fashion. In addition, initialization that is local to a single module is handled by declaring a static object in that module and performing the necessary initialization in the constructor of that object. We planned on per-module finalization in the corresponding destructors. However, a bug in Microsoft Visual C++ (present up to at least Version 5) causes such destructors not to run for modules in a DLL, so we were forced to eliminate any finalization code.

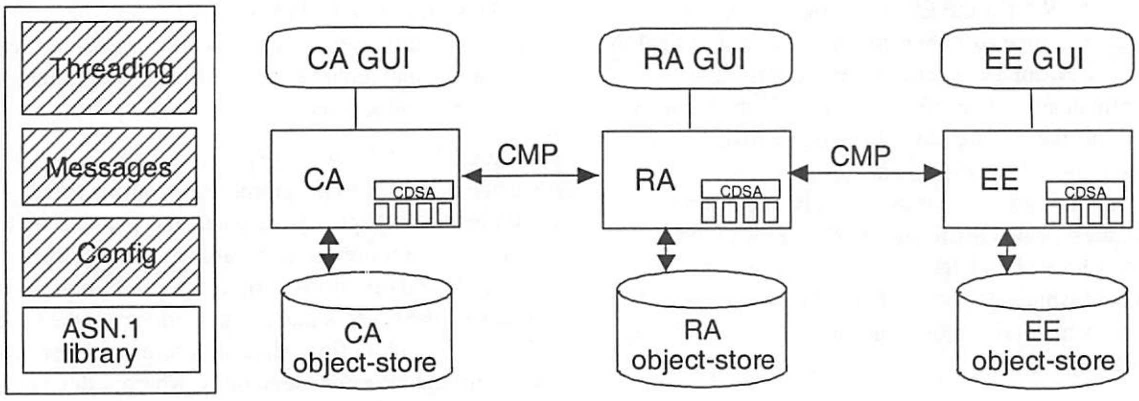


Figure 1. Jonah Architecture

We decided to expose the CDSA Trust Policy (TP) API as the *end application* API for Jonah (the API to evaluate chains of certificates, see Section 3), limiting the Jonah-specific API to certificate high-level life-cycle management functions (JNH API). Given the *unified-server architecture*, the entire JNH API is available on all servers. Services that are not appropriate at all servers (for example, the EE or RA cannot sign certificates) will return a "Wrong Server" status if invoked on an inappropriate server.

The majority of the API consists of *accessor* functions. These are routines that extract or modify individual fields of an object in the object-cache. Before invoking an accessor function, the API caller must first reserve the target object, which both locks the object against access by other threads and, if necessary, reads it into the object-cache from the object-store. After a series of accessor calls, the object may either be stored back in the object-store, or discarded, and unlocked. This mechanism allows for some degree of transactional behavior when updating object-store objects. The caller makes a series of changes to an object, then saves all the changes back to the object-store in a single operation. Most of the other API functions transfer an object from one server to another, by forwarding, fulfilling or rejecting a request. For example, the CA invokes a JNH_Create_Certificate routine to create a signed certificate. The routine creates a certificate and places it back in the object-store, marking the object-store entry complete, so that the certificate will be returned on the next poll from the RA.

The transaction model means that objects will sometimes appear and disappear in a server's object-store independent of any explicit user action. Any Jonah

GUI should be able to present a dynamic view of the contents of the server's object-store, so that certificate and revocation requests can be directly manipulated by the user. This leads naturally to an event-notification mechanism across the JNH API, whereby the back-end sends events to the GUI whenever an object-store object changes state. That state consists of several distinct pieces of information, encoded in a 32-bit integer value: an actual state value, a field that indicates whether an error has been reported or whether a password or PIN is required, a flag that says whether the GUI or the back-end currently "owns" the object, and a flag that indicates whether the object is active or a surrogate. On startup, the GUI makes a JNH call to request that the back-end send a series of events describing the current state of all object-store objects, allowing it to initially populate its view of the object-store.

2.5 Timers and Persistent Storage

CMP over TCP is a polling protocol, which requires a timer component at the RA and EE (the CA never initiates any communication that requires a delayed response). Whenever a message is sent for which a delayed response is expected, the sending server (RA or EE) will record the "next poll time" in the appropriate object-store entry, and queue a *timer* event to occur at that time for the object. On startup, a server walks through its object-store, and requeues timer events for any surrogate objects it finds, according to their next poll time. When the event fires, the corresponding object-store entry is used to poll the CA or RA to see if the delayed response is ready If not, another timer event is queued. In addition to polling, the CA must create CRLs on a regular basis, and both the RA and CA must handle key rollover. The *scheduler* saves queued events on disk, so that they will automatically be delivered on time if the server is running, or otherwise the first time the server runs after the scheduled time.

Creation of CRLs also requires that the CA maintain a secured on-line history of the certificates that it has issued. In Jonah, this history is stored in the *Issued Certificate List*, or ICL. The ICL contains a copy of each certificate issued, indexed by serial number. Since the index is implemented as a general-purpose skip-list [Pugh90], additional indices may be added reasonably easily. Since the ICL is mostly write-only (certificates are never changed, and can only be removed from the ICL by a pruning operation that deletes contiguous blocks of expired certificates from the start of the list), maintenance of these additional indices is relatively straightforward. In the future, we might want to provide a way for verifiers to ask the CA directly about the current status of a given certificate (for example, to support the PKIX Online Certificate Status Protocol).

On the CA, we need to store counters to maintain the serial numbers that will be used for certificates and CRLs. In addition, the CA and RA need a place to store their own certificates, and certificates for other RAs and CAs they trust. To satisfy these needs, we created a binary bin, a catch-all place where any persistent binary data is placed. The BinBin has a simple structure. The three serial numbers (for certificates, end-user CRLs and authority CRLs) appear first, followed by a BER-encoded sequence of certificates. The file is read in on server startup and cached in memory. Any modifications to serial numbers or certificates are written back to the file immediately.

3 Trust Policy (TP)

The Trust Policy (TP) module is implemented as a CDSA add-in (see Section 2.3). The TP interfaces are intended to be called directly by PKIX-based applications that build and validate certificates chains. Both of these activities are expected to be common functions of PKIX certificate using systems, like SSL [FKK96] and S/MIME [S/MIME]. The Jonah TP can be used for both the chain building and chain validation operations, via the CDSA CSSM interfaces *TP_CertGroupConstruct* and *TP_CertGroupVerify*, respectively. The other CDSA TP interfaces are not implemented. The TP performs minimal caching of data retrieved from the directory. Because of time constraints, the freeware TP does not cache directory data. For high performance applications like SSL which cannot afford to perform LDAP queries for each session, this is an unacceptable limitation. Further work is needed to create a trusted cache of certificates and CRLs already retrieved from LDAP and verified. The cache needs to age-out objects from the cache as they expire. Also needed would be options for applications to disable or flush the cache on demand, as well as set the age-out parameter (e.g., "all objects in the cache older than 1 day should be flushed, regardless of expiration time"). Some applications may prefer no cache, and trust only the latest data from the directory. The Jonah cache is not persistent across API invocations nor shared by running threads.

3.1 Validating Certificate Chains

TP_CertGroupVerify is used to validate PKIX certificate chains and uses the algorithm suggested in Section 6 of [RFC2459]. One exception is the validation algorithm for policy constraints and policy mapping, which was taken from [X.50997]. In our opinion, the [X50997] algorithm for validating the complex interaction among the certificate policies, policy mapping, policy constraints extension is more clearly explained and more rigorous that the algorithm defined in [RFC2459]. The X.509 algorithm also allows the user finer control over the initial validation state (via the initial-explicit-policy indicator, initial-policy-mapping-inhibit indicator, and initial-policy-set variables), unlike the [RFC1459] algorithm. Therefore, the Jonah TP validation algorithm is a faithful implementation of Section 6 of [RFC2459], with the exception of policy validation, which follows Section 12 of [X50997]. The inputs to chain validation, passed via parameters, are:

- A list of certificates ordered from an anchor certificate to the end-entity in question. How the chain is obtained is left unspecified in [RFC2459]. For example, the chain may have been sent as part of some application-level protocol flow. Alternatively, the caller may have previously constructed the chain using the TP interface, *TP_CertGroupConstruct*.
- The validation time for which the chain is to be evaluated. Usually, this is the current time, but one could compute the validity of a chain at some point in the past (with an extension to find necessary archival CRLs).
- The *initial-policy-set*, zero or more certificate policyIdentifiers, acceptable to the caller (e.g., a caller may choose to only accept paths consistent with some high-assurance certificate policy identifier).
- Two boolean flags, *initial-explicit-policy* and *initial-policy-mapping-inhibit*. These flags are identical in meaning to the fields in the PolicyConstraints extension [RFC2459]. The effect is to allow the caller to set the policy constraints state immediately, without waiting for the extension to appear in the path. For example, the user can set *inhibit-policy-mapping* to *true* to make sure the identifiers in the *initial-policy-set* are not mapped by a CA in the chain. The user can set *initial-explicit-policy* to *true* to ensure the critical certificate policies in the chain are consistent with the *initial-policy-set*.
- An optional pointer to an LDAP directory server. This pointer will be used, if necessary (and allowed by TP initialization parameters (see Section 3.3)), to retrieve CRLs to check certificate revocation status. The TP uses the JDL interfaces discussed in section 2.3.
- A pointer to a CSP to perform signature verification. The TP uses the JKL interfaces discussed in Section 2.3.

It is the caller's responsibility to verify the signature and revocation status of the top anchor certificate. It is assumed that the calling application maintains a list of trusted anchor certificates, which are deemed acceptable terminal points for certificate chains. Other implementations are possible (e.g., the TP could be configured with its own internal list of anchor certificates or trusted public keys). The application of the PKIX validation rules is straightforward. The Jonah TP performs the following checks:

- Name constraints: full support for permitted and excluded subtrees applied to Distinguished Names and alternative names for strings types rfc822Name, dNSName, directoryName, and uniformResourceIdentifier. Support for the legacy PKCS-9 emailAddress attribute in DNs and name constraint enforcement. The ASN C++ classes provide strong support for such character comparison.
- Policies: full support for certificate policies, policy constraints, and policy mapping.
- Revocation checking: for each CA in the chain, except the first, check issuer's *authorityRevocationList* (ARL) directory attribute for revocation status. For the end-entity certificate, check issuer's *certificationRevocationList* (CRL) directory attribute for revocation status. Other revocation checking methods, like OCSP, delta-CRLs, and CRLDistributionPoints, are not supported in the freeware TP.

As suggested by [RFC2459] and [X.509], the TP returns a list of evidence, or proof, collected as part of the validation process. The proof contains details of any policy mapping that occurred, a list of CRLs/ARLs used, and the list of certificate policyIdentifiers agreed upon by the CAs in the chain as acceptable. As an example, if validation fails because of revocation, the calling application might present the CRL in a GUI display. If validation succeeds, the caller still might decide to reject the certificate based on the policyIdentifiers returned.

3.2 Building Certificate Chains

Consider an application X, with a set of trusted anchor certificates, that needs to authenticate user Y. Authentication is possible if X can discover a certification path from Y to at least one of its trusted anchor certificates. This is expected to be a common operation needed by PKIX-enabled applications and is supported by the TP interface *TP_CertGroupConstruct*.

The PKIX LDAP schema [LDAP98] defines the *certificatePair* directory attribute, which stores all the cross-certificates issued to and by the CA. It holds both forward-certificates (certificates issued to the CA), and optionally, reverse-certificates (certificates issued by the CA). Since Jonah builds chains from the bottom up, only the forward element of the *certificatePair* is used. The Jonah TP chain-building algorithm is as follows:

1. For certificate Y, check if the issuer's name matches the subject name on one of the trusted anchor certificates T. If so, check that T in fact created certificate Y (by checking Y's signature). If so, stop, and return certification path {Y, T}. Otherwise, continue to step 2.
2. For certificate Y, retrieve all of issuer's forward certificates, F. Since Y's issuer may have multiple key pairs and multiple certificates for the same key, perform the following filtering steps to avoid following unpromising paths: (A) Discard any forward certificates whose subject name does not match Y's issuer name. This protects against bogus certificates in the directory and ensures proper DN name chaining. (B) If Y contains an *AuthorityKeyIdentifier* extension with the *keyIdentifier* field, discard any forward certificates that contain a non-matching *SubjectKeyIdentifier* extension. This makes sure F's certified public key is the same one used to sign certificate Y.
3. From the remaining list of filtered forward certificates F, check if any were issued by an anchor certificate T. If so, check that T in fact created F's signature. If so, stop, and return certification path {Y, F, T}. Otherwise, for each forward certificate F, recursively apply step 2.

The algorithm stops when (1) a certification path is found from Y to one of the anchor certificates T, (2) the search fails to find a path connecting certificate Y to one of trusted anchor certificate T, or (3) the search depth in the directory exceeds the INI defined *MaximumChainSearchDepth* (default value is 15). This value sets a reasonable limit on how much effort to expend building a certification path, and provides some protection against a directory seeded with bogus certificates.

Several important points were glossed over in the discussion above. Jonah implements a breadth-first search of the directory. As a result, the shortest chain is returned to the caller. Another possible algorithm is a depth-first search of the directory, which might return a different chain than a breadth-first search. For performance reasons, a depth-first search was avoided because of concerns over lengthy, recursive searches through the directory. Other algorithms were not tried; therefore our comments about one algorithm being to slow are speculative. We really did not have enough sample data to properly weight the merits of each approach. We expect to tune the TP caching model and directory search method based on real world scenarios.

In addition, there is no guarantee that the first chain found is in fact valid. Any of the intermediate certificates in the chain could be revoked, expired, invalid, or incorrectly signed. For performance reasons these checks are not done when building the chain. In fact, some of them cannot be done since extensions like NameConstraints and PolicyConstraints can only be checked from the top down. If the shortest chain happens to be invalid, returning it is of little value. We considered returning all chains (or all chains at a given depth) or returning only those chains that can be verified by *TP_CertGroupVerify*. These options require a search of the directory for all possible paths and were avoided for performance reasons. They also force the caller to decide which chain is the best. We decided to return the first chain that can be verified by *TP_CertGroupVerify* as the best compromise between accuracy and performance. Note that Jonah assigns each chain equal weight; it has no concept of intra-domain or inter-domain paths. All certificates in the *certificatePair* attribute are treated equally. Product versions of Jonah may apply additional rules that rank chains based on some criteria (like closeness to local name hierarchy).

3.3 TP Initialization File Parameters

The Trust Policy has configuration parameters that allow applications and users to set more lenient policy on verifying chains of certificates. The default behavior of the TP is to check the revocation of every certificate in a path. An option allows you to disable CRL checking. This may be useful in environments where a directory service is not used, or CRLs are not issued, or the revocation status is tracked external to the TP. When checking revocation, the TP finds the most relevant CRL available (usually the one with the highest CRLNumber). Two flags allow CRLs with *nextUpdate* dates in the past or with *thisUpdate* dates in the future to be considered valid. The default behavior of the TP is to reject them. Because of possible clock skew between different computer systems, CRLs issued on one system may not be active on another system until a few seconds or minutes in the future. Another flag allows a zero search result for a CA's CRL/ARL to be considered non-fatal. Every CA should have at least one

CRL and ARL, even if they are empty. Since PKIX requires every CA to issue at least an empty CRL, enabling this is not recommended. The default TP behavior is to reject chains where CRLs cannot be located. This flag might be useful in cases where a CA publishes certificates to the directory, but for some reason, does not publish CRLs.

Flags allow certificates with *notAfter* and validity dates in the past or *notBefore* validity dates in the future to be considered valid. The default TP behavior is to reject them. The reasons for allowing them are the same as for future CRLs. Turning on these flags turns off certificate validity checking, which is not PKIX compliant. Another flag allows name constraints to be applied only to the end-entity certificate. During validation, name constraints are enforced for every certificate in the chain. Some practitioners believe that, in practice, the two algorithms should give the same results for any reasonable hierarchy of CAs, and that any differences would not be important.

4 Smart Card Usage in Jonah

Smart cards are portable cryptographic devices that are suitable for storing certificates and keys, as well as performing cryptographic operations with the keys without releasing the private key off the card (specifically signing). Jonah contains a virtual smart card (one implemented totally in software with private keys and certificates stored in a file) which allows experimental use of Jonah in environments without smart card hardware. The EE, RA and CA servers each have a smart card configured. The CA and RA each initialize their smart cards to have a private key and corresponding certificate that are self-signed. This allows RAs to sign requests and CAs to sign CRLs without exposing their private keys. PKCS#11 [PKCS11] provides the interface to a smart card.

Smart card support performs the following functions: smart card initialization, changing a smart card password, storing and retrieving a certificate, storing a private key, retrieving information about a private key, returning information about the certificates associated with a private key, generating a key pair, signing, and verification. CAs and RAs use it as a storage mechanism if they need to distribute their certificate to other entities. EEs also store certificates for verifiers with whom they exchange data, although they are limited by the available storage on the smart card. Typically, a smart card will include its own software package used to administer the smart card for functions such as initialization and password management. Administration of a PKCS#11 smart card was somewhat problematic. This is because while the PKCS#11 standard has administrative functions, most smart cards provide their own non-standard administration interfaces. While we provide functions to administer the virtual smart card through PKCS#11, administration of other smart cards will certainly be specific to each smart card.

The Jonah smart card interface provides an API for smart card operations to the rest of Jonah. It translates Jonah ASN.1 data structures and semantics into CDSA data structures and semantics, then calls CDSA to access the smart card. Use of the CDSA framework in our virtual smart card support provides consistency across Jonah for access to our cryptographic providers. We anticipate that CDSA will make it easier for Jonah to use PKCS #11 compliant hardware smart cards. The cryptographic operations provided by the virtual smart card go back through the CDSA interfaces to CSPs that are configured to work with Jonah. Since smart cards devices typically work through a serial connection at speeds around 9600 baud, and smart card cryptographic functions are fairly slow, we do not think this layering will cause any notable performance problems.

The smart card CDSA support translates calls into PKCS#11 calls. In the Jonah reference implementation, a PKCS#11 virtual smart card is used. A smart card must support PKCS#11 functions for key pair generation, data storage, and signing to be fully functional for the Jonah reference implementation. Private keys are stored on a PKCS#11 smart card by splitting the private keys into their base parts, including a modulus. When keys are generated on the smart card, a key identifier is created for indexing keys on the smart card. In addition, the public key is returned to Jonah for use in a certificate request. When using the smart card for key pair generation, the private key will usually never leave the smart card. This is the safest way to generate and store private keys. If archival of keys is required, the keys can be generated in software and archived, then stored on the smart card. We have struggled with issues around protecting private keys. The best design for storing and protecting keys for archival purposes seems to be to use a secret key to protect the private keys while stored in a PKCS#12 [PKCS12] file. Hardware smart cards may not be willing to export private keys they generate. There may be later interoperability problems between such smart cards and applications that insist on having private keys in their own local keyring. We also need to protect virtual smart card data stored in a file. We settled on utilizing hashes of the security officer and user pins, as well as combinations of random and secret keys.

We discovered several issues while integrating virtual smart card support with the Jonah public key infrastructure. We found several places needing translation layers between what Jonah needs and what smart cards traditionally provide. As discussed above, we needed several translation layers to allow access to the smart card via CDSA. In addition, the general practice with smart cards is to use the index generated by the card when referencing items stored on the card. Jonah needs to index and manage multiple certificates that correspond to a single private key. For example, a CA may have different certificates for certificate signing and for CRL signing, but both may reference the same private key.

5 Jonah's Graphical User Interface

The Jonah GUI, like the rest of Jonah, must run on multiple platforms including Windows NT and various platforms of UNIX (specifically Solaris and AIX). As freeware, the GUI is downloaded and built on all of the above platforms. Staffing issues and the initial snapshot schedule meant that from concept to initial release, the GUI had to handle certificate requests and communicate with the back-end within 3 months. This section describes how we met those goals, as well as issues encountered so far.

While the portability layer (see Section 2.2) allowed us to write the back-end in a portable, platform-neutral fashion in plain C++, this approach is not suitable for GUI implementation. We decided to build the GUIs in Java [Java], taking advantage of its platform independence, language safety, and object-oriented classes. The Java Development Kit (JDK [Borl98a]) 1.1, with its compiler and run-time virtual machine, is free and is supported on all of our target platforms. JDK 1.2 was not considered because of lack of support for UNIX platforms like AIX. The GUI code invokes the JNH API via a Java Native Interface (JNI [Cay98]) wrapper layer. We used the Java Foundation Classes (JFC)/Swing classes [Nels98], and associated freeware widgets, instead of the Java Active Window Toolkit (AWT [Flan97]) to save development time. We chose Borland JBuilder 2.0 as our Java development environment because it produces 100% pure cross-platform code, has a fully supported implementation of the JFC, has an upgradeable path to future releases of JDK 1.2 and beyond, shares its Interface Developer's Environment with Delphi (a mature environment for building Pascal programs), and our team members have had excellent experience with Borland products in the past. We used JBuilder's ability to create individual freely releasable widgets as Java Beans for Jonah-specific inputs like a spin button for dates and X500 distinguished name input. The Jonah GUI consists of three Jonah GUI executables for the CA, RA and EE. This Java byte code is zipped up as JAR [Borl98b] files into four packages, CA, RA, EE and Base. The Base class, like the back-end, is used for the common functionality shared by the servers. This code sharing has already eased debugging and support. Use of inheritance within the OO paradigm also paid off in reusing code. The layering provided by the JNI wrapper made it very easy to divide up the work and to detect which layer crashes were in.

The event-notification mechanism required some effort to integrate with a Java GUI. Java threads are not necessarily implemented using the platform's native threading mechanism. A Java Virtual Machine (JVM [Deit97]) may multiplex a single native thread to service all Java threads [Davi96]. Therefore, a native thread does not necessarily have sufficient thread context data to be able to invoke Java code. Java threads may invoke native routines, but the JNI does not permit calls in the reverse direction. This restriction required that the Java GUI implement an event collection thread, whose sole purpose is to check for events and pass them back to the main Java thread. The JNI wrapper layer implements a message-queue to pass event data from the native notification call-back routine to the event collection thread. This, in turn, required that the Jonah locking primitives be exposed at the JNH layer, so that the event notification callback routine and the Java event collection thread could use them to synchronize access to the message queue. Since the JVM might be implemented as a single native thread, we could not block the collection thread while it waited for an event, since this might block the entire JVM. Therefore, if the collection routine finds no events waiting, it calls back into Java and performs a Java `sleep()` operation before checking the message queue again. Thus, apart from very brief periods when the collection thread is holding a Jonah mutex to inspect the contents of the message queue, it is using pure Java synchronization mechanisms, which allow the JVM to continue to run regardless of how it implements Java threads. We added a second similar notification mechanism to allow the back-end to send text messages for display to the user, either on the GUI status-bar, as pop-up dialog boxes, or in a scrolling log window. The last was extremely useful during debugging. The current GUI has three application threads: the main thread that maintains the GUI itself, and two event collection threads for events and text messages.

One difficulty was providing a front-end that was user-friendly and intuitive to the PKIX standard back-end, supporting only standard data types for displaying information. The PKIX standard does not always provide the information one would choose while designing a front-end. An example is validity dates. The PKIX standard uses start and end dates for certificate validity. An end-user or RA administrator might more reasonably enter a validity duration to be applied against the time the CA signs the certificate or a start time and a duration. Extra code and checks are required in the GUI to set up the data as PKIX standard. Extra code keeps the user's notion of duration aligned with the beginning and ending validity dates. Distinguished Name (DN) support is also a challenge. The GUI is coded to understand the full standard format of DNs. It contains a full encoding of the attribute types and their ordering for the X.500 useful object classes, and can lead the user through the creation of a DN by presenting the next valid attribute type as a field for editing. This Java bean is fully table driven and can therefore be easily extended to support additional schema. Although the entire team has had several discussions of how best to support DN input and editing, we are not thoroughly satisfied with our current solution. In addition, expertise in GUI development and in security and PKIX standards is split across team members. Every new GUI feature generates a great deal of discussion on both sides of the split. While this is a substantial investment in time, the fresh ideas on both sides have produced many breakthroughs. Having a single contractor develop the entire GUI reduced coordination and code overhead, and expertise also guaranteed that the front-end followed standard Windows and Motif behavior, although it necessitated many long hours of work.

6 Porting Jonah

This section discusses the build environment, Standard Template Libraries (STL); Java portability; issues between NT and UNIX; and CDSA, message, and threads porting issues. The back-end, C++ portion of Jonah was originally built using the MKS make program [MKS97] with batch file wrappers to simulate the OSF Development Environment (ODE [ODE]). Both the NT batch files and the MKS Makefiles were not portable, since the MKS syntax is similar to but not identical to systems provided by the AT&T UNIX operating system. Our Java build environment, JBuilder, is also an NT-specific tool. In order to port all the build procedures to UNIX platforms, we used ODE. ODE has been ported to many platforms including UNIX, NT and Windows. We have not found a software compilation environment that supports as many platforms as the OSF ODE environment supports. It supports both C++ and Java.

One area that turned out to be a problem in porting Jonah from NT to AIX was the Standard Template Libraries. The Standard Template Library [Step94] provides a set of C++ container classes and generic algorithms. It is based on research in generic programming and generic software libraries. When the template is instantiated or invoked, types are supplied as parameters and methods are created for those types. The Standard Template Library has been adopted as part of the February 1998 ANSI/ISO C++ standard. Microsoft Visual C++ conforms to this version of the standard. Unfortunately, most compiler venders have not had time to come out with conforming implementations. The C++ compiler that we use on AIX is based on a 1992 version of the ANSI/ISO C++ standard. To allow programmers to use the STL on platforms that do not have a compiler that conforms to the February 1998 version of C++ standard, STL has been heavily parameterized with conditional compilation flags that indicate whether a compiler supports various template features. For example, our AIX compiler does not support default arguments for templates, so we had to back out of using that feature. It takes some investment of time to determine what template features a compiler supports and how to correctly set the corresponding flags. We may have similar problems for any new platform Jonah is ported to. In addition, the AIX compiler also tends to be very strict about what it allows. For instance, it requires that if a template class is defined then each method and operator used by the class must be defined, even if the application does not invoke it. The Microsoft Visual C++ 5.0 compiler only requires the methods and operators be defined if they are actually instantiated.

There was very little work to port the Java GUI to AIX. We wrote some build rules for the ODE build environment that compile the Java source and create a JAR for each of the GUIs, and wrote a simple script that invoked the GUIs under the JDK. No changes to the Java source code were needed. By adhering to the JDK's JNI standard, the Java code is able to call the native C++ code in a portable manner. In some cases, native data types were incorrectly declared (for instance, as "long" instead of "jlong"). This generated some mapping bugs. The Java code was easier to port than the C++ code, though based on efficiency of generated code, we believe coding the back-end in C++ was the right decision.

We ran into some problems with general differences between NT and UNIX. Since NT is case insensitive, our developers were not careful when applying case to new file names and then maintaining case sensitive #include statements. Since the UNIX operating system is case sensitive, there were several places where we had to either change the name of the include file or the #include directive in the code. Also, the initialization files contain information about where to store various Jonah files in the file system. On NT, you include the drive specifier along with the path, while on UNIX systems there is not a drive specifier. These directives need to be modified to conform to the file system structure of each operating system. Our CDSA code contains routines for storing information about where the plugins should be found and loaded. On the NT platform, this information is stored in the NT registry as the default database to store OS-specific information. On AIX, the code stores the data in the AIX Object Data Management (ODM) database, which performs an equivalent function. The code to find plugins needs to be changed for each operating system to use a data store similar to the NT registry or the AIX ODM. One possible option is to use a DB44 database [Slee] if the OS does not have a common database.

Jonah attempts to segregate all of its platform-specific code to a single layer, called the OSSRV layer. Exceptions to this are found in the CDSA layer and a few places where small changes are conditionally compiled depending on the platform. The OSSRV layer consists of support for the portability library (see Section 2.2). We had one porting decision to make around message files, and found one porting problem in threads. Instead of converting the error messages to the format that the native AIX message tools require, and creating calls to AIX messaging routines, we decided to port the message catalog routines used on NT to AIX. The Jonah code supplies an XPG4 like set of routines and a message catalog generating program (gencat). Since the Jonah code only opens one catalog file and most gencat routines expect to open a separate catalog for each .msg file, we ported the gencat and XPG4 routines provided with the Jonah delivery. In addition, we ran into a problem where the default stack size on AIX for each thread (100K) was not large enough for the C++ code. This problem caused Jonah to crash in a number of random ways. We were surprised to find that we needed to increase the default stack size of the threads created by Jonah to 256K.

7 Testing

The final aspect of Jonah development is testing. As discussed by Marick [Mari97], the role of a test team is to find bugs. However, there is a distinction to be drawn between a test designed to find a large volume of bugs with minimal significance and one designed to find bugs that will have a major impact on the core use of the code. The goal of the Jonah test team is the latter. This was defined as those bugs that users encounter during basic use of the code, including usability problems impacting the user's ability to understand how to use the code. The focus of the test is, therefore, on exercising the main paths of the Jonah code (certificate creation, certificate revocation, and issuing CRLs). While many of the challenges with testing Jonah are familiar territory, some are exacerbated by, or unique to, its status as a freeware package. This section discusses the issues involved in testing the Jonah freeware code. It describes differences between freeware testing and regular product testing, setting priorities, and the challenges the test team faced.

Since Jonah is a freeware package that uses PKIX drafts as requirements and specifications documents, the test team believed that test processes needed to be modified. In the usual process, testing is divided into Functional Verification Testing (FVT) and System Verification Testing (SVT) [IBM98]. FVT verifies that the code functions correctly with respect to product specifications. FVT exercises the external and internal interfaces, functions, error handling capabilities, and the maintainability and serviceability of the product. SVT verifies that the code works correctly as a *system*. The goal of SVT is to test products in a customer-like environment (at times, utilizing industry customer scenarios) thus ensuring delivery of high-quality solutions that meet or exceed customer expectations. SVT exercises scenarios that test not only product requirements, but also load and stress; interoperability and coexistence; installation, migration, configuration and connectivity; reliability, system-level error recovery, and internationalization. For Jonah, with a goal of finding the most important defects, the test team executed a mixture of traditional FVT and SVT scenarios. We exercised the external interfaces (using the GUI and some APIs) and functions using an integrated system. We attempted to exercise the product requirements (using the PKIX drafts as the specifications), as well as product installation using minimal configuration scenarios.

Another aspect of testing freeware code is that limited resources (hardware, people, and time) are allocated to the effort. As a result, the test team is much smaller than usual. Thus, we had to set priorities. As previously discussed, the test team chose to focus mainly on the mainline paths through the code, using defaults instead of making major modification to different parameters. Additionally, since we understood that the first potential customers (IBM product groups) would use the APIs to exercise the Jonah code, the test team chose to spend time using the API set. This also allowed us to begin building our test suites, which could be used for later product testing. For the API testing, we chose the C interface since we had both example code and a skill set in place to code C applications. In respect to LDAP testing, we limited the test to one LDAP server, choosing the IBM LDAP 2.0 server [IBMLDAP], since this was the easiest server for us to install. We also developed a process to document the important bugs requiring fixing.

The biggest testing challenge was the test team's lack of knowledge about the technology. Because of organizational changes, within days of receiving the Jonah test assignment, management expected a positive contribution from the test team. Learning the technology involved reading the Internet drafts, talking with the developers, and practicing with the code at every opportunity, and asking numerous questions. Contributing to this challenge was the fact that the Jonah code was itself lacking documentation and continually referenced the PKIX drafts. Finally, the test team came from a very structured test environment and was used to testing mature products. Jonah required us to adjust our thinking from a strict process driven environment (including test plan execution plan, daily status, and war room meetings) to a less structured environment that focused on delivering the code as soon as possible. Our test team has worked hard to make the adjustment and they have contributed substantially to the quality of our code.

8 Status and Conclusions

The Jonah code is being hosted by the Massachusetts Institute of Technology (MIT), who assume responsibility for code distribution and change control. It is downloadable from the web site at MIT at http://web.mit.edu/pfl. MIT agreed to host the code because they have hosted security code with export control issues in the past (such as Kerberos), and because Jeff Schiller is both IETF Security Area Director and is responsible for the MIT network. The Internet Mail Consortium hosts the Jonah discussion list at http://www.imc.org/imc-pfl/. Also, interest in PKIX reference implementations is increasing. NIST [NIST] and DSTC [Oscar] have also announced reference implementations.

Interoperability is important to our goal of being a reference implementation. We are currently involved in a sequence of interoperability testing events with Entrust, NIST, Xeti, and Baltimore Technologies. We have exchanged certificate requests and responses to test interoperability. This has uncovered both mistakes in the different implementations and deficiencies in the PKIX specifications. The majority of the problems discovered in the first round of testing were with the ASN.1 encoding rules. The CMP specification uses ASN.1 explicit tagging by default. The CRMF specification uses ASN.1 implicit tagging by default. ASN.1 also has rules about when you must use explicit tagging. For instance, explicit tagging is required within a *choice*, which overrides the default of implicit tagging in the CRMF specification. Another related problem is that some of the ASN.1 structures have changed from earlier drafts to the final RFC. The lack of change control in versions of these RFCs means that small changes are often not discovered until interoperability testing. In addition, the specification is vague about how to associate a certificate request with the password for that request shared with the RA. The purpose of the interoperability events is to identify problem areas and feed then into the next version of the protocol. Participants will be reporting on the results at the July IETF meeting.

One of the largest challenges we have faced implementing freeware reference code has been organizational. Our team crosses organizational units, company cultures, geographical sites, and time zones. We have worked hard to work together, and have been recognized for it with two IBM teamwork awards. Many of the organizational difficulties we had were exacerbated when working across expertise or geography. Even though our business case had been made, we had continuing difficulty getting and keeping resources in the face of pressure in nearby revenue-making groups who also needed resources. With a small team, some of whom were juggling other commitments, we adopted a heads-down, lets-code attitude without design documents or a detailed schedule. This has produced the desired aggressively timed freeware snapshots, but also placed an added burden on areas such as GUI development, testing, porting, and management. For example, testing has not always been told just what was in a code drop, which made it difficult for them to determine where to concentrate their efforts. They are also located in Texas, while the team leads work out of Massachusetts, so special effort to communicate was required. As we transition internally from freeware to internal product support, we are producing more documents and communicating more effectively.

We are not aware of many other papers on the topic of developing and distributing freeware reference implementations. Kerberos [SNS88] is a well documented standard backed by freeware code. However, most discussion of freeware implementations occurs on email lists or informally at IETF sessions. We hope others will document their systems and experience implementing freeware reference implementations.

Acknowledgments

Jonah would not have been possible without the contributions of Les Kendall, Rina Rivera, Shiu Fun Poon, Maryann Hondo, Dave Dyer, Charlie Kaufman, Mark Davis, and Stacey Powers. This paper has benefited from the work of our editor, Lynda Balthorp, the comments from our anonymous reviewers, and the direction of our shepherd, Carl Ellison.

Bibliography

[ASN.1]"Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation", ISO.IEC 824-1.

[Borl98a] "Borland JBuilder2. Developers Guide." Borland International Inc., Scotts Valley, CA 95067-0001, 1998.

[Borl98b] "Borland JBuilder2. Quick Start." Borland International Inc., Scotts Valley, CA 95067-0001, 1998.

[BSAFE] "RSA BSAFE CryptoC." http://www.rsa.com/rsa/products/cryptoc/index.html.

[Cay98] Cay S. Horstmann, Gary Cornell, "Core Java Volume II- Advanced features." Sun Microsystems Press, 901 San Antonio Road, Palo Alto, California, 1998.

[CDSA97] "Common Data Security Architecture." Intel, http://developer.intel.com/ial/security/specifications.htm.

[CMC98] Michael Myers, Xiaoyi Liu, Barbara Fox, and Jeff Weinstein. "Certificate Management Messages over CMS." November 11, 1998, http://www.ietf.org/internet-drafts/draft-ietf-pkix-cmc-04.txt.

[Davi96] Stephen R. Davis, "Learn Java Now." Microsoft Press, 1996.

[Deit97] H. M. Deitel, P.J. Deitel, "Java How To Program." Prentice Hall Inc., Upper Saddle River, New Jersey, 1997.

[FKK96] A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 Protocol." Netscape Communications Corp., Nov 18, 1996.

[Flan97] David Flanagan, "Java In A Nutshell." O'Reilley & Associates Inc., 101 Morris Street, Sebastopol, CA 95472, May 1997.

[IBM98] IBM, "IBM Development Guidelines (DevGuide) Integrated Product Development (IPD) Guide Release 1.7", 1998, http://w3.enterlib.ibm.com/cgi-bin/bookmgr/books/zdgipd.

[IBMLDAP] IBM, "Lightweight Directory Access Protocol." http://www.software.ibm.com/network/directory.

[ISO8825] "Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER)." ISO/IEC 8825-1.

[ISO9945] "Information technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]." ISO/IEC 9945-1:1996.

[Java] "Java Platform Documentation." http://java.sun.com/docs/.

[LDAP98] Sharon Boeyen, Tim Howes, and Pat Richard, "Internet X.509 Public Key Infrastructure LDAPv2 Schema." September 1998, http://www.ietf.org/internet-drafts/draft-ietf-pkix-ldapv2-schema-02.txt.

[Mari97] Marick, Brian, "Classic Testing Mistakes." Testing Foundations, 1997. http://www.stlabs.com/marick/Classic/mistakes.html .

[MKS97] "MKS Toolkit User Guide." Mortice Kern Systems Inc., 35 King Street North, Waterloo, Ontario N2J 2W9, Canada, 1997, http://www.mks.com/solution/tk/.

[Nels98] Mathew T. Nelson, "Java Foundation Classes." McGraw-Hill Inc., 11 West 19th Street, New York, NY 10011, 1998.

[NIST] "MISPC CD-ROM Request Form." http://csrc.nist.gov/pki/mispc/refimp/cdj2.htm.

[ODE] "ODE Home Page." http://www.ede.com/ode/.

[Oscar] "Oscar: DSTC's Public Key Infrastructure Project." http://oscar.dstc.qut.edu.au/.

[PKCS5] "Password Based Cryptography Standard." RSA LABS, 3rd draft, 2/2/1999. ftp://ftp.rsa.com/pub/pkcs/pkcs-5v2/pkcs-5v2draft3.pdf.

[PKCS7] "PKCS #7: Cryptographic Message Syntax Standard." RSA Laboratories Technical Note, Version 1.5, Revised November 1, 1993, ftp://ftp.rsa.com/pub/pkcs/ascii/pkcs-7.asc.

[PKCS10] "PKCS #10: Certification Request Syntax Standard." RSA Laboratories Technical Note, Version 1.0, November 1, 1993, ftp://ftp.rsa.com/pub/pkcs/ascii/pkcs-10.asc.

[PKCS11] "PKCS#11: Cryptographic Token Interface Standard." RSA Laboratories, Redwood City, CA. http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-11.html.

[PKCS12] "PKCS #12: Personal Information Exchange Syntax Standard." RSA Laboratories Technical Note, Version 1.0, April 30, 1997, ftp://ftp.rsa.com/pub/pkcs/pkcs-12/PKCS12.PDF.

[PKIX] "Public Key Infrastructure (X.509) Charter." http://www.ietf.org/html.charters/pkix-charter.html.

[Pugh90] William Push, "Skip lists: a probabilistic alternative to balanced trees." Communications of the ACM, Vol. 33, No. 6 (June 1990).

[RFC2459] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile." January, 1999, ftp://ftp.isi.edu/in-notes/rfc2459.txt.

[RFC2510] C. Adams, S. Farrell, "Internet X.509 Public Key Infrastructure Certificate Management Protocols." March 1999, ftp://ftp.isi.edu/in-notes/rfc2510.txt.

[RFC2511] M. Myers, C. Adams, D. Solo, and D. Kemp, "Internet X.509 Certificate Request Message Format." March 1999, ftp://ftp.isi.edu/in-notes/rfc2511.txt.

[Slee] "The Sleepycat Software Home page." http://www.sleepycat.com/.

[S/MIME] Blake Ramsdell, Editor, "S/MIME Version 3 Certificate Handling." http://www.imc.org/draft-ietf-smime-cert, Expires June 14, 1999.

[SNS88] J.G. Steiner, B.C. Newman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems." Proc. USENIX Conference, February 1988.

[Step94] Alexander Stepanov, Meng Lee, "The Standard Template Library." HP Labs Technical Report HPL-94-34 (R. 1), August 1994.

[UNIX] "The Single UNIX Specification - 5 Vol Set for UNIX 95." The Open Group, T910.

[X.50997] "ITU-T Recommendation X.509: The Directory Authentication Framework." 1997.

Scalable Access Control for Distributed Object Systems

Daniel F. Sterne (dan_sterne@nai.com)
Gregg W. Tally (gregg_tally@nai.com)
C. Durward McDonell (durward_mcdonell@nai.com)
David L. Sherman (david_sherman@nai.com)
David L. Sames (david_sames@nai.com)
Pierre X. Pasturel (pierre_pasturel@nai.com)
NAI Labs, Network Associates, Inc.

E. John Sebes (ejs@securify.com)
Kroll-O'Gara Information Security Group

This research was funded by DARPA under contract F30602-97-C-0268.
Approved for Public Release, Distribution Unlimited.

Abstract

A key obstacle to the widespread use of distributed object oriented systems is the lack of scalable access control mechanisms. It is often necessary to control access to individual objects and methods. In large systems, however, these can be so numerous that the resulting proliferation of access control information becomes overwhelming. We describe Object Oriented Domain and Type Enforcement (OO-DTE), a technology for organizing, specifying, and enforcing access control that has been prototyped and integrated with commercial ORBs and SSL. OO-DTE provides fine-grained control and scalability via a compilable symbolic policy language. We discuss our experience building and using OO-DTE and compare OO-DTE with the access control terminology, concepts, and requirements described in CORBA Security.

1. Introduction

Although distributed object technology has matured greatly during recent years, the lack of practical, integrated security mechanisms remains an obstacle to its use in many application domains. CORBA[1] and Java Remote Method Invocation (RMI)[2] are two of the more popular distributed object technologies. Version 1.0 of the CORBA Security specification (CORBASec) [CORB] was initially released in January 1996. However, commercial product releases that fully comply with its Level 2 (general-purpose) requirements are just beginning to emerge. Moreover, even Level 2-compliant products may prove inadequate because Level 2 requirements in some functional areas are neither complete nor sufficiently stringent. In particular, CORBA Security does not establish requirements for security management mechanisms. These mechanisms can have a major impact on the usability and scalability of the implementation. Java security, as represented by Sun's Java 2 and the recently announced Java Authentication and Activation Service (JAAS), is still undergoing rapid evolution. Moreover, although Sun has released plans and a draft specification for the RMI Security Extension [RMISEC], the facilities currently provided for controlling RMI-based access by clients to remote Java servers are very limited.

Providing practical access control mechanisms for distributed objects - whether based on CORBA or Java RMI - is a challenge because the necessary characteristics for such mechanisms are not well understood. One particular problem stems from the common requirement to control access to individual objects and methods, even in systems where the number of these abstractions is enormous. For example, consider the CORBA naming service, which allows CORBA objects to be bound to portions of CORBA's hierarchical name space and provides queries about name bindings. It would be unwise to give every user complete freedom to use the naming service, i.e., unrestricted ability to *bind* any object to any part of the name space, or *unbind* any name-to-object binding.

[1] Common Object Request Broker Architecture from the Object Management Group (OMG)
[2] Java is a product of Sun Microsystems.

Mechanisms are needed to allow a user to invoke these naming service methods, but only on appropriate objects. Unfortunately, attempting to enumerate separate access control attributes for every method of every object in a large Object Oriented (OO) system causes an overwhelming proliferation of access control information that results in a loss of understandability, manageability, and ultimately, security. This problem is worsened when Access Control Lists (ACL) are used as security attributes [OSG] [DENG] [HU] because each ACL can contain many entries, including some that appear contradictory. Moreover, the combined effect of ACLs can only be understood by applying complex precedence and ordering rules [BALD] [DOWN].

This paper describes the results of a DARPA-funded research project to develop access control technology for distributed objects. By access control technology, we mean mechanisms for 1) specifying the sets of objects and methods that users and automated processes may access, and 2) protecting objects and methods from unauthorized access. The technology we have developed is called Object Oriented Domain and Type Enforcement (OO-DTE). OO-DTE is an outgrowth of our earlier research into access controls for secure operating system kernels [BADG] [WALK] [SHER].

OO-DTE was developed with the following goals:

- Object Oriented - It should support OO abstractions and take advantage of OO relationships, especially interface inheritance.
- Scalability and Manageability - It must be suitable for controlling access in large distributed applications systems comprising thousands of processes, objects, and methods.
- Fine-grained Control - It must allow access to be controlled at the level of individual objects and individual methods.
- Role-based Access Control (RBAC) - It should support access rules organized according to user roles (job titles or functional responsibilities) rather than user identities or security clearances.
- Compatibility with Commercial Products – It should be designed as a plug-in module that can be linked into commercial object request brokers (ORBs) and distributed object infrastructure components.
- Transparency - It should support *security unaware* applications, i.e., applications that do not explicitly invoke access control or other security services.

This paper is organized as follows: Section 2 provides background, terminology, and motivation. Section 3 describes DTEL++, the compilable language used in OO-DTE to specify access control policies. Section 4 describes the implementation, features, and status of our OO-DTE prototypes. Section 5 compares OO-DTE to CORBA Security concepts and terminology. Observations on our experience with OO-DTE, including preliminary performance data, are discussed in Section 6. Section 7 provides a summary.

2. Background

OO-DTE extends Domain and Type Enforcement (DTE) concepts to distributed object systems. DTE is a set of access control mechanisms for UNIX kernels [BADG] [WALK] [SHER]. DTE was, in turn, based on Type Enforcement, as proposed originally by Boebert and Kain [BOEB] in 1985. For historical reasons, DTE and OO-DTE have continued to use terminology that originated from Type Enforcement even though those terms are now overloaded with additional meanings.

On a DTE operating system, a security attribute called a *type* is associated with every file, directory, device, and IP packet. Types are assigned to these system resources according to a site-specific policy. Types are typically assigned to indicate the kind of information contained, its sensitivity, integrity, or origin. A security attribute called a *domain* is associated with every process. Domains are assigned to processes to indicate the kind of computing tasks they are intended to perform and to prevent unnecessary access to system resources. Each domain is defined as a collection of access rights. Each right is expressed as the ability to access information of a specified type in one or more access modes, e.g., read, write, execute, send, receive, etc.

One of the innovations of DTE was the use of a compilable high-level language to define types and domains. This language is called the DTE Language or *DTEL*. DTEL was also used to assign types to the file hierarchy via a set of general rules with exceptions. This allowed an entire file system, potentially consisting of millions of files, to be "labeled" in a concise and understandable manner by means of a few DTEL statements. This contrasts with ordinary UNIX systems that require an administrator to examine the permission bits on vast numbers of files and directories *individually* in order to infer how files of different sensitivities are arranged and distributed throughout the file hierarchy.

OO-DTE is an attempt to extend DTE notions to distributed object-oriented systems. We have concentrated initially on developing OO-DTE for CORBA-based systems but have plans to adapt OO-DTE for Java RMI. OO-DTE includes an extended policy language called DTEL++ that provides constructs for assigning types to methods. DTEL++ also includes rules for propagating default type assignments from modules and interfaces to enclosed methods[3], and for propagating type assignments to inherited methods. DTEL++ defines two new access modes for methods: *invoke* (needed by clients) and *implement* (needed by servers).

The first OO-DTE prototype ran on a DTE kernel and exploited the DTE kernel's access control facilities for interprocess communications [SHER]. That prototype, sometimes referred to as *Kernel Level OO-DTE*, involved using and modifying the Inter Language Unification (ILU) ORB[4]. These modifications were needed to make ILU use the DTE kernel's extended *send* and *receive* system calls, which transmit type attributes with messages and prevent senders and receivers from accessing unauthorized information types.

Although the DTE kernel provides important security advantages, there is little interest outside the research community in non-standard kernels. To broaden the potential audience for our research we have subsequently developed newer prototypes called *Above Kernel OO-DTE*. Above Kernel OO-DTE was specifically designed to run on mainstream commercial operating systems and is the primary focus of this paper.

2.1. Domains, Roles, and Role Authorization

OO-DTE has been designed to support role-based security policies, that is, policies that grant users access rights according to their assigned roles (duties) within an organization [NICO]. In our earlier DTE kernel operating systems, each role was represented as a collection of domains. This allowed each task initiated by a user to run as a separate process in a "small domain" providing only the minimum set of access rights needed to accomplish that task. This approach was developed in accordance with the principle of least privilege [SALT]. OO-DTE, however, is designed to run on mainstream operating systems whose kernels lack comparable process spawning and access control facilities. Consequently, in OO-DTE, each role is represented by a single domain; hence, the terms *role and domain have become virtually synonymous.*

In OO-DTE, a user may be authorized for multiple roles, but each connection associated with a user process is bound to a single role (domain). OO-DTE roles can be constructed hierarchically by defining domains in terms of other domains, in the manner of Baldwin's Named Protection Domains [BALD]. OO-DTE does not provide facilities for dynamic or static separation of duty; unlike some practitioners [KUHN], we regard these as distinct from the fundamental aspects of role-based access control.

Roles, domains, and types provide several levels of indirection in the authorization policy. As discussed later, users hold X.509 certificates that contain an OO-DTE domain as a privilege attribute. The policy defines domains in terms of their access rights to types. The policy also assigns types to methods. Through these levels of indirection, a user's authority to invoke a method can be determined. Figure 1 shows the relationships between users, domains, types, and methods:

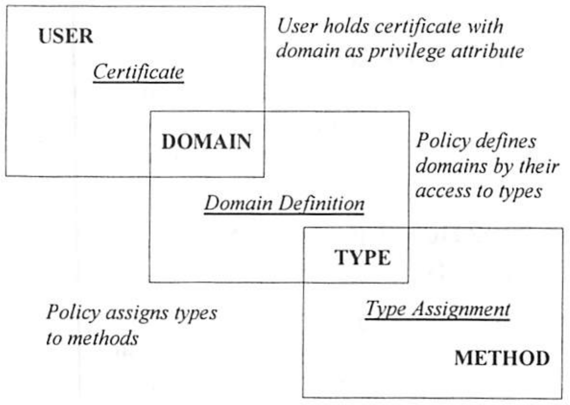

USER
Certificate
User holds certificate with domain as privilege attribute
DOMAIN
Policy defines domains by their access to types
Domain Definition
TYPE
Policy assigns types to methods
Type Assignment
METHOD

Figure 1 - Users, Roles, and Authorization

3. DTEL++ Policy Language

DTEL++ is the policy language used to specify OO-DTE security policies [TALL]. It is used to declare types, to assign a type to each method that might be invoked, and to define domains in terms of the types each domain can invoke or implement. We will motivate the discussion of the policy language by illustrating how it can be used to control a simple application system.

[3] Note that CORBA uses the terms "interface" and "operation" in place of "class" and "method". We use both sets of terms interchangeably.

[4] ILU was developed by Xerox Corporation, Palo Alto Research Center.

3.1. Sample Application

Suppose we are writing an application to manage the books in a library. The primary objects in our application are librarians, patrons, and books. One can enter, remove, and find books in the card catalog, reserve books, and check books in and out. The Client-server interface for this application, as specified in CORBA's Interface Definition Language (IDL), might look like this:

```
module Library {
  struct BookDescription {...};
  interface Patron {...};
  interface PatronDatabase {...};

  interface Book {
   readonly attribute BookDescription desc;
   void checkOut (in Patron patron);
   void checkIn ();
   long numberAvailable();
   long numberReservations();
   void reserve (in Patron patron);
   };

  interface BookDatabase {
    Book newBook (in BookDescription desc,
                  in long copies);
    void removeBook (in Book book);
    Book findByTitle   (in string title);
    Book findByAuthor  (in string author);
    Book findBySubject (in string subject);
  };
};
```

In our policy, we want to allow both librarians and patrons to find books in the card catalog and to reserve books. Additionally, we want to allow only librarians to check books in and out, to enter new books into the catalog, and to remove books from the catalog.

3.2. Type Definitions and Simple Type Assignments

Every DTEL++ policy must declare types. These types are then used to label interface methods so that access to the methods can be controlled. In our policy we declare two types, `safe_t` and `restricted_t`[5] with the OO_type statement:

```
OO_type   safe_t, restricted_t;
```

Methods that both patrons and librarians are allowed to invoke will be assigned the `safe_t` type, and methods that only librarians are allowed to invoke will be assigned `restricted_t`.

Each method in each interface must be assigned a type. This can happen in one of several ways. The most straightforward way is with an explicit assign statement for the method. Since patrons can safely be allowed to look up books in the catalog, the type `safe_t` is assigned to the method `findByTitle()` by the assign statement:

```
assign   safe_t   findByTitle;
```

A second way of assigning types to methods is to assign a default type to the methods within an interface or module, as in:

```
assign   restricted_t _DEFAULT;
```

Each module and interface may be assigned a default type. By declaring a default type of `restricted_t`, any method that does not receive a type via an explicit assign statement, as shown previously, will receive the type `restricted_t`. This allows us to construct our policy so that patrons can invoke only those methods that have been explicitly granted to them.

Other ways of implicitly assigning types will be discussed in Section 3.5.

3.3. Domain Definitions

A DTEL++ policy must define the permissions granted to each domain. For each type declared in the policy, a given domain can have `invoke` permission, `implement` permission, neither permission, or both permissions. Only those permissions explicitly stated in the domain definition are granted to the domain. In our example application, we can allow patrons to perform safe operations, and no others, with the domain declaration:

```
domain patron_d = (invoke->safe_t);
```

We permit librarians to perform both safe and restricted operations with:

```
domain librarian_d =(invoke->safe_t,
                             restricted_t);
```

Domains can be concatenated. Since the `librarian_d` domain has a superset of the privileges of the `patron_d` domain, it could have been defined as:

```
domain librarian_d = patron_d,
                     (invoke->restricted_t);
```

While the impact upon our simple example is minimal, had the definition of the `patron_d` domain been substantially more complex, the above construction of the `librarian_d` domain would still allow the policy to show very clearly the relationship of the two domains. This allows one to express larger policies in a

[5] We have adopted the convention that the suffixes "_t" and "_d" denote type and domain identifiers

more concise, and more understandable way, for example, policies in which roles are defined hierarchically.

The software that maintains the library database must implement all of the methods and requires the following domain definition:

```
domain server_d = (implement->safe_t,
                               restricted_t);
```

For an example of both invoke and implement permission, consider augmenting the server_d domain. We may want to give it the invoke privilege for the safe_t type if it has to invoke some of the listed methods internally to do its job. We would re-define the server domain statement as this:

```
domain server_d = (invoke->safe_t),
        (implement->safe_t, restricted_t);
```

3.4. Sample Policy

Here is the DTEL++ policy for our library system:

```
OO_type safe_t, restricted_t;

module Library {

  assign restricted_t DEFAULT;

  interface Book {
    assign safe_t { _get_desc,
                    numberAvailable,
                    numberReservations, reserve
                    };
  };

  interface BookDatabase {
    assign safe_t { findByTitle,
                    findByAuthor,
                    findBySubject };
  };
};

domain patron_d    = (invoke->safe_t);
domain librarian_d = (invoke->safe_t,
                      restricted_t);
domain server_d    = (implement->safe_t,
                      restricted_t);
```

The first line declares the two types that we are using. Next, we open the Library module and assign restricted_t as the default type for all methods in the module. Then we open the Book and BookDatabase interfaces and explicitly assign the safe_t type to methods that may be invoked by patrons. Finally, we define our domains: applications run by patrons will run in the patron_d domain and be allowed to invoke only safe_t methods; librarian applications will run in the librarian_d domain and be allowed to invoke safe_t and restricted_t methods; and any servers will run in the server_d domain and be allowed to implement both safe_t and restricted_t methods.

When an assign statement for a default type appears within the scope of an interface statement, the default type applies to the methods of that interface. When such an assign statement appears outside the scope of an interface statement, but within the scope of the module statement, the default type is accepted as the default type by all interfaces within the module that do not contain their own default assign statement to override it.

In our example, we assigned a default type of restricted_t for the entire Library module. All interfaces within the module will have restricted_t as their default type since none of our interfaces contains an assign statement to reset the default type within the interface. All methods in the Patron and PatronDatabase interfaces, which may be numerous, need not be mentioned in the policy; they will receive the restricted_t type by default. In the Book interface we assigned the type safe_t to some of the methods, but since we did not explicitly assign types for the checkIn() and checkOut() methods, they both receive the interface default of restricted_t.

3.5. More Complex Type Assignments

Inheritance

By default, DTEL++ method types are assigned recursively from base interfaces to derived interfaces. For example, assume an interface ChildrensBook that derives from the Book interface described above. The ChildrensBook::checkOut() method would automatically receive the same type, restricted_t, that was assigned to the checkOut() method in the Book interface. It is also possible to explicitly assign a type to the ChildrensBook::checkOut() method so that the inherited type is overridden.

The inheritance of type assignments in DTEL++ can be fine-tuned through the use of the **-i**, **-l**, and **-f** flags. These flags appear immediately after the keyword assign and modify the effect of the type assignments on inherited methods or derived interfaces. These are discussed further in [TALL].

Per Object Access Control

DTEL++ allows different objects having the same interface to be assigned different access control characteristics, but requires that such objects be bound to names like those defined by the CORBA Naming Service. Objects with different names can then be treated differently.

For example, there may be some books in a library that no one may check out, perhaps because they are antiques. Instead of having to define a new `AntiqueBook` interface, we can leave the IDL for the application unchanged and add the following lines to our DTEL++ policy:

```
// No one has permission to invoke null_t
OO_type null_t;
...
module Library {
  ...
  template AntiqueBook : interface Book {
    assign null_t checkOut;
  };
  assign AntiqueBook /Books/Antique/;
  ...
};
```

The collection of type assignments for a given interface is called its *default type template*. This includes both explicit and implicit type assignments received through inheritance relationships or default types. An object that is not named has the *default type template* applied to it. The example above defines a *named type template* `AntiqueBook`. The *named type template* applies only to objects with names that fall under the namespace assigned to the type template (/Books/Antique/).

Object names are arranged in a directory-like structure, and are given to the objects by security aware object factories. In our example, an antique book should be bound to a name such as "/Books/Antique/1003" (for book #1003 in the catalog, for example), while a regular book should be bound to a name such as "/Books/1351" (or possibly no name at all). The following DTEL++ line says that any object with a name that starts with "/Books/Antique/" should have its methods typed as in the `AntiqueBook` template, rather than the standard (unnamed) `Book` template.

```
assign AntiqueBook /Books/Antique/;
```

4. Implementation

Several OO-DTE prototypes have been built. This paper focuses on two more recent prototypes that have been built as plug-in modules for two commercial ORBS running on mainstream operating systems: Orbix (Iona Technologies) on Solaris and Visibroker (Inprise) on Windows NT.

4.1. DTEL++ Compiler

The DTEL++ compiler uses DTEL++ policy files and associated CORBA IDL files as input to generate data files used by an OO-DTE ORB at runtime. The compiler output consists of 1) tables that bind types to methods, and 2) domain definitions that are described in terms of the types each domain can invoke or implement. The compiler compares the module, interface, and method identifiers in a DTEL++ policy with their counterparts in the IDL files and reports inconsistencies. If a method, interface, or module appears in the DTEL++, but not in the IDL, it is reported as an error. However, a method may legally appear in IDL and not in the DTEL++. In this situation, the compiler uses the type assignments provided by inheritance and default types to infer the type of the method.

4.2. Run-time Architecture

Like other ORBs, both Orbix and Visibroker provide a "plug-in" interface for extending the ORB's functionality. The CORBA specification calls these plug-ins *interceptors*. Although a standard interceptor interface is under development by the OMG, current interceptors for ORBs are vendor-specific.

The primary runtime components of our Orbix and Visibroker OO-DTE prototypes are interceptor-like plug-in components that reside in both the client and server ORBS. The behavior of these plug-in components is driven by the data files produced by compiling DTEL++ policies. The plug-in in the server's ORB protects the server. For each operation request it receives, it determines whether the requesting client is authorized to *invoke* the requested operation; if not, it rejects the request and sends a CORBA:NO_PERMISSION exception to the client. The plug-in in the client's ORB protects the client. For each operation, the client ORB determines whether the server is authorized to *implement* the operation (i.e., provide the service); if not, the ORB rejects the request, thereby preventing it from being sent to the server. This prevents the client from inadvertent interactions with any malicious applications that attempt to impersonate servers.

OO-DTE access control plug-ins rely on vendor-provided Secure Socket Layer (SSL) [DIERK] packages to authenticate clients and servers and protect traffic between them. SSL was selected as the communications security mechanism for OO-DTE, primarily because it is the most widely supported security technology for CORBA.

The access control plug-ins also rely on SSL to convey to servers the authorization of clients, and to clients the authorization of servers. SSL uses X.509 certificates that are issued to each principal. Each X.509 certificate used with OO-DTE contains the principal's domain.[6] Consequently, a user authorized to act in multiple roles will have multiple certificates, each containing a different domain, and must designate the certificate appropriate to the role in which he or she is currently acting. This is analogous to a retail customer choosing from his or her wallet a credit card appropriate to (i.e., accepted by) the merchant with whom a purchase is sought.

SSL handshaking is performed whenever a client and server establish a connection. The handshaking process causes exchange and cryptographic verification of information contained in the certificates including the domains of the client and server. Each ORB retains this information for the lifetime of the connection and makes it accessible to the access control plug-in for use in authorization checking as described above.

Figure 2 shows the relationships among the OO-DTE components:

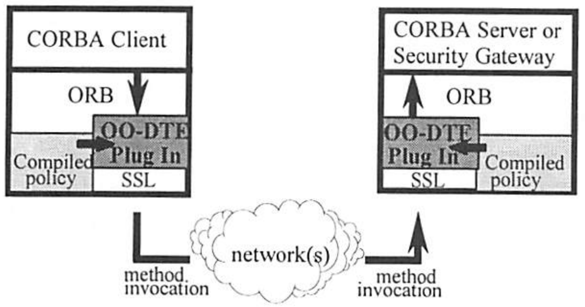


Figure 2 -OO-DTE Components

4.3. Per-Object Access Control

The OO-DTE plug-in makes access control decisions based on the OO-DTE type of the operation, the domain of the client (or server), and the definition of that domain as described in the DTEL++ policy. The type assignment for the operation is also described in the policy, as discussed in Section 3 above. The type depends on the operation, interface, and, when per-object access control policies are used, the object's name, e.g., /Books/Antique/1003. The operation is part of the request object passed to the plug-in. The interface name can be determined by invoking local methods on the target object, such as retrieving the Repository ID. Obtaining the object name, if present, required the development of additional, non-standard mechanisms.

[6] In the initial prototype of Above Kernel OO-DTE, the Organizational Unit of the Distinguished Name contains the domain information.

Per-object access control relies on assigning unique identifiers (object names) to objects and organizing the object names in a hierarchical manner. CORBA deliberately avoids providing unique identifiers for objects, so implementing OO-DTE required inventing a specialized naming mechanism. There are several requirements for this mechanism:

- An object can have, at most, one OO-DTE name.
- There must be a means for the plug-in to determine the name of an object given a reference to the object, preferably without making a remote invocation.
- The names must be organized in a hierarchical manner to allow easy association with type templates.
- Object names must be assigned at object creation and thereafter be immutable. Objects cannot be named after creation.

An obvious candidate for naming objects is the CORBA Naming Service. However, the Naming Service does not provide a two-way association between objects and names. Given a name, it is possible to find the associated object using a Name Server. However, it is not possible to determine an object's name from a reference to the object. Furthermore, CORBA Naming provides for a many-to-one mapping between names and objects.

The OO-DTE prototypes use a combination of mechanisms to assign names to objects. First, the object name is bound to the object in a secure implementation of the Naming Service. The Naming Service provides structure to the naming hierarchy and ensures that object names are not re-used. Second, a 'stringified' form of the object name is stored in the *object key* portion of the object reference to provide the two-way association between the name and object. This allows the object name to be extracted from the object reference by the OO-DTE plug-in on the client side. Since object factories are responsible for creating and naming the object correctly, per-object access control requires a security-aware application. However, the impact is limited to the factory.

4.4. Policy Distribution and Synchronization

OO-DTE takes a decentralized approach to authorization decisions in order to keep policy information close to OO-DTE enforcement mechanisms, which are resident in client and server plug-ins, and provide faster access decisions than could be obtained by consulting with a centralized access decision server. With this approach, however, all policy changes must potentially be propagated to multiple hosts to ensure consistent policy enforcement. The OO-DTE prototype provides a central point of control for policy changes and distribution. Policy update notifications are pushed from this point to individual OO-DTE hosts, providing loose synchronization of policy changes.

OO-DTE policy distribution uses CORBA methods for policy change notification and for transport of the policy updates. A *master policy server* accesses the authoritative copy of the policy. *Local policy servers* on each host register with the master policy server at startup. Applications (client and server programs) register with their host's local policy server at startup. When the policy administrator changes the master policy, an event notification propagates from the master policy server to each of the registered local policy servers, and then to the security plug-ins in the client and server ORBs. Local policy servers invoke methods on the master policy server to retrieve the updated policy and then copy it into local files, which are then read by the plug-ins of local client and server processes. In this way, policy changes are put into effect transparently to applications and 'on-the-fly', without stopping or restarting applications. For scaling to a large number of policy subscribers, policy distribution servers can be arranged into a hierarchy, with each server communicating policy changes only with its immediate superior and subordinates. All policy notifications and transfer operations are protected by SSL.

4.5. ORB Gateway and Multi-Protocol Object Gateway

OO-DTE components have also been integrated into two network security gateways: the *ORB Gateway* and its successor, the *Multi-Protocol Object Gateway*. Both Gateways are deployed on or behind firewalls where they intercept CORBA's Internet Inter-ORB Protocol (IIOP) traffic addressed to servers in the enclave. The Gateways perform control access in accordance with a DTEL++ policy before forwarding requests to the servers. If a Gateway denies a request, the request is not forwarded. Neither Gateway requires modifications to CORBA object references or IIOP messages. In the event that a remote client cannot present an X.509 certificate containing an explicit DTE domain, the Gateways can be configured to use other information about the client, including the client's IP addresses or the contents of other X.509 fields, to infer or synthesize a domain for use in access control decisions. Once a domain is established for that client, the rest of the access decision process proceeds as described above. The Multi-Protocol Object Gateway provides the same functions as the ORB Gateway but supports a subset of Java RMI in addition to CORBA IIOP and will be a focus of our future research.

5. Comparison to Access Control in CORBA Security

In this section we relate OO-DTE to the access control terminology and features described in CORBASec [CORB].

5.1. OO-DTE Types vs. Standard Rights Family

According to the CORBASec model, an *access policy* translates privilege attributes held by a principal (e.g., group membership) into *rights* and specifies which rights a principal must hold to invoke operations on CORBA objects; these are called *required rights*. Rights belong to *rights families*, which are simply sets of rights, like sets of access modes. CORBASec defines a *standard rights family* that consists of the rights "get", "set", and "manage". These correspond to read-only, update, and administrative access modes, respectively. CORBASec allows other rights families to be used, but discourages them.

While the standard rights family (get, set, manage) is simple and general, its usefulness seems quite limited. To illustrate, we will attempt to use it to protect the `Book` interface in the library application discussed in Section 3. Consider the requirement that patrons and librarians are allowed to reserve books, but only librarians are allowed to check out books. Both of the `Book` methods `reserve` and `checkOut` change the application's state. Hence, for both, the most appropriate required right in the standard rights family is "set". But if the "set" right is granted to both librarians and patrons, patrons will be allowed, inappropriately, to invoke the `checkOut` method. On the other hand, if only librarians are granted the "set" right, then patrons will be prevented, inappropriately, from invoking the `reserve` method. Unfortunately, the standard rights family cannot readily address these simple and ordinary requirements.

The underlying problem is that the interface to an object often includes multiple "set" methods that update different parts of an object's state and should be accessible to different groups of principals. To adequately specify the required rights for an interface, a *distinct* right might be needed for each update method. Hence, the optimal number of update rights depends on the application and cannot usefully be reduced to the single "set" right provided by the standard rights family. By contrast, the OO-DTE "rights family" is simply the collection of types declared by the DTEL++ policy writer to suit the characteristics of the application, e.g., safe_t and restricted_t. The simplicity of the OO-DTE solution for the library application presented above illustrates OO-DTE's flexibility and effectiveness at solving this common problem.

5.2. Per-Object Access Control

OO-DTE provides named objects and type templates so that access control can be configured on a per-object basis when needed. CORBASec states, however, that "Required Rights are characteristics of interfaces, *not* instances. All instances of an interface, therefore, will always have the same Required Rights."[7] Consequently, it is not possible to directly specify per-object access control using CORBASec's Required Rights mechanism. To get around this restriction, the application architect is forced to put individual objects or collections of objects – all of which belong to the same interface – into separate *Security Policy Domains*. The specification defines a Security Policy Domain as "a set of objects to which a security policy applies for a set of security related activities and is administered by a security authority."[8]

Consider the use of per-object access controls to provide special handling of antique books in our library example. Addressing this handling requirement by using the CORBASec Required Rights necessitates placing antique books into a different Security Policy Domain than ordinary books. This seems awkward and artificial because both kinds of books are objects "to which a [single] security policy applies" and both are probably controlled and "administered by a [single] security authority". These quotes from CORBASec suggest that both kinds of books *should* belong to the same Policy Domain. Moreover, the administrative and programming ramifications of using multiple Security Policy Domains are unknown because CORBASec provides no guidance on 1) how Policy Domains are created, deleted, and configured, and 2) how objects are moved among domains. The concepts and implementation issues of Security Policy Domains are now being re-examined by the CORBASec community. An RFP seeking clarification of the issues surrounding Policy Domains is currently underway. For simple applications like our library application, forcing designers to use multiple Policy Domains seems cumbersome, risky, and unnecessary given that OO-DTE achieves the desired result without them.

[7] See [CORB], Section 15.6.4, "Access Policies"
[8] See [CORB], Section 15.3.8, "Domains"

5.3. Supporting and Exploiting Inheritance

DTEL++ uses a collection of wild-card rules to facilitate the assignment of types to methods and make these assignments as compact and intuitive as possible. In particular, DTEL++ exploits the inheritance hierarchy. When types are assigned to the methods of a base interface, by default, the inherited methods in all derived interfaces automatically inherit those type assignments. As a result, when derived interfaces are added to an application, few if any additional DTEL++ statements may be required, even if the base interface was complex and required numerous DTEL++ statements. In addition, DTEL++ provides a variety of optional flags to control and selectively override the propagation of type assignments through the inheritance hierarchy.

In the examples in CORBASec, Required Rights are shown as a table that enumerates the rights required for each method in *each interface*. Inheritance is not discussed. This suggests that when a derived interface is defined, it may be necessary to enumerate the required rights for each inherited method even if they are identical to the rights for the base interface's methods. This will be burdensome when deriving new interfaces from lengthy base interfaces. Furthermore, it is clearly unnecessary, as DTEL++ illustrates, and is conducive to errors and inconsistencies. The specification doesn't prohibit an implementation from exploiting inheritance to make the enumeration of rights more compact. But, it also fails to provide guidance or insights into the benefits, issues, or features that might be required.

6. Experience

The OO-DTE features described above have been implemented on three different ORBs (ILU, Orbix, and Visibroker) and three different operating systems

(DTE-enhanced BSD/OS, Solaris, and Windows NT) and in network security gateways. Although we have not tested OO-DTE with an operational CORBA application, we have used several non-trivial demonstration applications, including three that were written by other organizations; one of these was many tens of thousands of source lines in length.

6.1. Performance – Preliminary Results

We have recently begun collecting and analyzing OO-DTE performance data. Our initial target for testing is our plug-in interceptor for the Visibroker ORB, Version 3.2. This ORB and our plug-in for it are written in Java and are intended for use with CORBA programs written in Java. In our tests, we timed a repetitive set of operation invocations provided by a demonstration application written by another DARPA contractor. Start and stop times were collected on the client machine issuing the invocations. The start time was captured prior to invoking the first operation in the set, but after establishing a (secure) connection to the server. The stop time was captured after completing the set of operation invocations.

The operation that produced the most consistent test results is called "add_annotation". It sends 18 (application-layer) bytes of ASCII "annotation" data to a server and requests that the server replace the current annotation associated with a particular CORBA object. Timing data was collected for four configurations of this application: 1) no security or plug-in components; 2) SSL alone; 3) SSL with a "null" plug-in on the client and server; and 4) SSL with the OO-DTE plug-in on the client and server. This version of Visibroker supports only one configuration of SSL that provides confidentiality of messages (encryption) as well as authentication and integrity. For these tests, the client machine was a 133 MHz Pentium with 32 MB RAM. The server machine was a 166 MHz Pentium with 64 MB RAM. The results shown in Figure 3 below are based on computing the incremental timing differences between these four configurations, and dividing the differences by 1000, the number of times the operation was repeated between timings. This calculation provides the average time contribution per operation per component. Note that these numbers represent the time consumed by the *combination* of the client and server working together in a sequential manner.

Component	Average time per operation
• Application	4.09 ms
• SSL	2.95 ms
• Null plug-in	1.0 ms – 1.25 ms
• OO-DTE access check	0.02 – 0.10 ms
Total (average)	8.33 ms

Figure 3 - Time contributions for the "add_annotation" operation

As shown, OO-DTE access checks, without any optimization, are relatively fast, consuming approximately 1% of the total time for this lightweight operation. Using the upper bound of 100 microseconds for a pair of OO-DTE checks (one on the client and one on the server) each check requires 50 microseconds or less on slow Pentiums (166 MHz or 133 MHz). With performance tuning, it is likely that OO-DTE speed could be improved, though it's not clear that it would matter much, given how slow the Visibroker plug-in mechanism and SSL (in this mode) are by comparison.

We also ran a number of other tests involving other operations, larger data transfers (and hence more encryption) and varying amounts of simulated compute time on the server. No other experiment provided consistent data that was at odds with the above.

6.2. Observations

In our testing, most OO-DTE features worked as expected. Some of the features that appeared to be unimportant in the design phase proved quite useful after implementation. For example, automated policy distribution proved invaluable during "policy debugging" because it allowed us to distribute experimental policy changes quickly to multiple hosts, including our security gateways. Other features designed to support larger, more complex policies have had limited use so far, e.g., inheritance rules among the type templates used for per-object access control. Nevertheless, we still expect them to prove valuable for operational systems.

Overall, DTEL++ has proven to be simple, flexible, and easy to use as a policy specification language. The

DTEL++ policies we have experimented with have tended to be compact, usually no larger than their associated IDL files, and in some cases, much smaller. This is largely due to the automatic inheritance of assigned type bindings and the ease of establishing default type bindings for an entire CORBA interface or module.

Our experience using DCE[9] integrated with Orbix Security to secure CORBA applications has heightened our appreciation of OO-DTE features. Compared with OO-DTE, we perceive DCE as a set of low-level enforcement mechanisms that lack appropriate abstraction facilities, especially for OO. (It might be possible, however, to hide DCE mechanisms under a layer of better administrative tools.) In particular, DCE does not help one *organize* an access control configuration as a set of high level *patterns,* as is required to enforce role-based policies and other organizationally mandated policies. For example, when a derived class is created that should have the same access control characteristics as its base class, a DCE administrator must create a new ACL for the derived class that is a duplicate of the base class's ACL. DCE provides no general way to specify that both ACLs are instances of a common pattern. For distributed OO applications that use inheritance, this makes DCE seem conducive to error and difficult to maintain, particularly when access configuration changes need to affect a collection of classes uniformly. Another advantage of OO-DTE over DCE is that OO-DTE facilitates inspection, audit, and analysis of the access control configuration because a complete description of the current access control configuration resides in a single or small set of DTEL++ files. By contrast, auditing a DCE configuration requires opening and inspecting every ACL in the system.

A few aspects of OO-DTE did not work as well as hoped. The syntax for defining type templates and the inheritance relationships between type templates is somewhat confusing and will be refined. We also plan to experiment with a syntax that allows multiple interfaces per type template. Some DTEL++ semantics have proven ambiguous or difficult to implement in the compiler, for example, the semantics governing the propagation of default type assignments into interfaces with multiple inheritance. A rule is needed that specifies which set of defaults (if any) propagates into the derived class. One possibility is to have no default in cases that could be ambiguous, and instead require an explicit type assignment.

[9] Distributed Computing Environment from The Open Group

It is desirable to detect as many DTEL++ errors as possible at compile time because it is generally more difficult to detect and diagnose errors at runtime. Consequently, the DTEL++ compiler verifies that all module, interface, and operation names in a DTEL++ policy match identifiers in the associated CORBA IDL files and reports any mismatches. However, we have experienced other sources of DTEL++ errors that could potentially be detected prior to runtime by additionally cross-checking. First, names of DTE domains that represent user roles could be compared against the set of DTE domains known to the certificate authority that issues the X.509 certificates used with OO-DTE; the names of DTE domains in these certificates should be consistent with the DTEL++ policy. Second, a DTEL++ policy that uses per-object access controls assigns type templates to names in CORBA's object name space. It might be possible to detect naming errors in a DTEL++ policy prior to runtime by consulting with the CORBA naming service to validate the existence, availability, ownership, etc. of these names.

Another area for improvement is display of DTEL++ policies. DTEL++ was designed as an adjunct to CORBA IDL. Consequently DTEL++ is terse and contains little information already present in IDL. For example, DTEL++ does not explicitly describe the inheritance hierarchy nor does it enumerate all the operations that are part of an interface. While avoiding redundancy with IDL simplifies maintenance, it also means that neither IDL nor DTEL++ by itself presents a "complete picture" for a human administrator. A useful addition to OO-DTE would be a viewer tool that provides an integrated view of IDL and DTEL++. For example, it might enumerate all the operations in the interface, and for each, the DTE type assigned as a net result of defaults and explicit overrides in the DTEL++ policy. In a related development, researchers at Secure Computing Corporation have developed a graphical administrative tool that can generate compilable DTEL++ [THOM].

DTEL++ was originally conceived of as a collection of annotations that would be embedded in IDL. These annotations would be either stripped out by a DTEL++ preprocessor prior to compiling the IDL or disguised as IDL comments. The approach described here was adopted instead because it allows an application's IDL and DTEL++ policy to be changed independently in many cases, thereby minimizing the "ripple effect". For example, a policy can be fine-tuned by adding new roles without changing the files containing IDL. This is desirable because without special tools, a change in IDL files might trigger unnecessary recompilation of

IDL, recompilation of the generated stubs and skeletons, and relinking of the client and server programs that use them.

OO-DTE does not provide any features for delegation, i.e., features that allow a client to dynamically authorize an intermediary server to act on its behalf when invoking methods on a target server. The primary motivation for delegation is to limit the extent to which an intermediary server must be trusted. This is accomplished by allowing individual clients to grant subsets of their own privileges temporarily to the intermediary. While this is intuitively appealing, it is not clear to the authors that delegation significantly reduces the amount of trust that must be placed in an intermediary server in many circumstances. For example, when the intermediary concurrently supports multiple clients having diverse privileges, the intermediary must be trusted to use the correct set of privileges for each request it makes on a target server. Furthermore, the cryptographic overhead and administrative costs of delegation appear significant. Moreover, in order for a client to explicitly authorize each target service used by the intermediary, the client must know in advance how the intermediary is implemented, i.e., must know what services the intermediary will use to implement the services it offers to clients. This seems to violate the well-established software design principles of abstraction and information hiding. Because the benefits of delegation do not seem to outweigh these costs, the authors have no plans at this time to support delegation in OO-DTE.

6.3. Future Plans

OO-DTE research and development is continuing under DARPA funding. We plan to continue research in the following areas:

- Attribute Certificates - For compatibility with SSL, our current implementation appropriates the Organizational Unit field of an X.509 (identity) certificate and uses it to represent a principal's role. It would be preferable to pass this information separately in an attribute certificate.
- Role Authorization Database - We are currently developing an alternative approach for exchanging role authorization information as part of the IIOP message stream and validating it. The goal is to reduce the need to issue and revoke certificates when authorization changes occur, which may be frequently. In this approach, each application's plug-in validates its counterpart's requested role by comparing it against a local role authorization database. The database is distributed and updated via the same mechanisms that are used to distribute OO-DTE policy files.
- Policy Modules - Our current approach to policy distribution causes all DTEL++ policy subscribers to receive all policy updates. It would be preferable for each subscriber to receive only the policy updates it needs. Supporting this will require mechanisms for organizing a DTEL++ policy as a collection of separable modules that can be subscribed to independently.
- Java RMI - We plan to extend OO-DTE for use with Java RMI. One issue that has surfaced is that RMI supports overloaded method names. Because CORBA IDL does not, neither does DTEL++. DTEL++ will need to be extended accordingly for RMI.
- Role Instances - In some applications, there is a need to distinguish among different instances of the same role and extend different rights to each. For example, in a medical records application, there may be several instances of the primary physician role. Each instance should have access to the same fields in patient records, but for different patients. We plan to extend OO-DTE to support such requirements.

7. Summary

Although distributed object technology has matured greatly during recent years, the lack of practical, integrated security mechanisms, particularly for access control, remains an obstacle to its deployment in many application domains. It is often necessary to control access to individual objects and methods. In large systems, however, these can be so numerous that the resulting proliferation of access control information can be overwhelming. We have described Object Oriented Domain and Type Enforcement (OO-DTE), a research technology for organizing, specifying, and enforcing access control that has been prototyped and integrated with commercial ORBs and SSL. OO-DTE is an outgrowth of our earlier research into access controls for secure operating systems.

Our experience developing OO-DTE suggests that it has been successful in meeting its original goals:

- Object Oriented - OO-DTE is object oriented. DTEL++, its compilable policy language, resembles CORBA IDL, supports OO abstractions, and takes advantage of the inheritance hierarchy to improve the conciseness and understandability of access control policies.

- Scalability and Manageability - DTEL++ provides wild-card techniques for assigning OO-DTE types to methods based on the inheritance, lexical name scoping, and object naming hierarchies. These techniques allow a few DTEL++ statements to control thousands of objects and methods. OO-DTE also provides mechanisms to automatically distribute policy changes to large numbers of clients and servers.
- Fine-grained Control - OO-DTE provides fine-grained control over individual objects via named type templates. Type templates allow distinct combinations of OO-DTE types to be assigned to individual objects or collections of objects whose names share subtrees of the object name space.
- Role-based Access Control - OO-DTE provides access control based on user roles. Roles, implemented as OO-DTE domains, can be defined in an application-specific manner via DTEL++. A user's authority to act in a role is currently represented by the value of a field in the user's X.509 certificate.
- Compatibility with Commercial Products - OO-DTE has been implemented as a plug-in module for Iona Orbix and Inprise Visibroker, the market leading ORBs.
- Transparency - The OO-DTE plug-in supports security-unaware applications by automatically invoking SSL authentication and access checks without direct participation by application code.

We have compared OO-DTE to the access control terminology and features of CORBASec and described what we believe are important advantages of OO-DTE over CORBASec's Required Rights and standard rights family. These advantages include the ability to 1) address a wider range of real-world access control requirements; 2) provide per-object access control within a single Security Policy Domain; and 3) express policies for derived classes in a more compact and understandable manner.

We have also identified several areas where OO-DTE could be improved or extended including adapting OO-DTE for use with Java RMI. Improvements and extensions of these kinds, combined with OO-DTE's designed-in scalability, should allow OO-DTE to mature over time from a research technology to a practical base of mechanisms suitable for commercial products and large-scale distributed object systems.

Bibliography

[BADG] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker. "*A Domain and Type Enforcement UNIX Prototype,*" USENIX Computing Systems Volume 9, Cambridge, MA, 1996.

[BALD] R. W. Baldwin, "*Naming and Grouping Privileges to Simplify Security Management in Large Databases*", Proceedings of the IEEE Symposium on Security and Privacy, page 116 (May 1990).

[BOEB] W. E. Boebert and R. Y. Kain, "*A Practical Alternative to Hierarchical Integrity Policies,*" Proceedings of the 8th National Computer Security Conference, pp. 18-27, Gaithersburg, MD, September 1985.

[CORB] CORBA Services: Common Object Services Specification, Chapter 15, Security Service Specification, November 1997 Object Management Group, Inc.

[DENG] R. H. Deng, S. K. Bohnsle, W. Wang, A. A. Lazar, "*Integrating Security in CORBA-Based Object Architectures*", 1995 IEEE Symposium on Security and Privacy, page 50.

[DOWN] D. Downs, J. Rub, K. Kung, C. Jordan, "*Issues in Discretionary Access Control*", 1985 IEEE Proceedings of the Symposium on Security and Privacy, page 208.

[HU] W. Hu, DCE Security Programming, 1995, O'Reilly & Associates, Inc.

[DIERK] T. Dierks, C. Allen, The TLS Protocol Version 1.0, Internet Engineering Task Force, RFC 2246, January, 1999.

[KUHN] D. Kuhn, J. Barkley, A. Cincotta, D. Ferraiolo, S. Gavrila, "*Role-Based Access Control for the World Wide Web*", 20th National Information Systems Security Conference, page 331.

[NICO] V. Nicomette and Y. Deswarte, *"An Authorization Scheme for Distributed Object Systems"*, 1997 IEEE Symposium on Security and Privacy, page 21.

[OSG] Orbix Security Guide, 1991, IONA Technologies PLC.

[RMISEC] Java RMI Security Extension, Early Look Draft, Sun Microsystems, Palo Alto CA, 1999 http://java.sun.com/products/jdk/rmi/rmi-security.pdf

[SALT] J. Saltzer and M. Schroeder, *"The Protection of Information in Computer Systems"*, IEEE Proceedings, 63(9), March 1975.

[SHER] D. Sherman, D. Sterne, L. Badger, S. Murphy, K. Walker, S. Haghighat, *"Controlling Network Communication with Domain and Type Enforcement"*, 18th National Information Systems Security Conference, October 1995, page 211.

[TALL] G. Tally, D. F. Sterne, C. D. McDonell, "Sigma Project: DTEL++ Language Specification", Trusted Information Systems, September 1998.

[THOM] D. Thomsen, R. C. O'Brien, J. Bogle. *"Role Based Access Control Framework for Network Enterprises,"* Proceedings of the 14th Annual Computer Security Applications Conference, pp. 50-58, December 1998.

[WALK] K. W. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. *Confining Root Programs with Domain and Type Enforcement.* Proceedings of the 6th USENIX Security Symposium, San Jose, CA, 1996.

Certificate-based Access Control for Widely Distributed Resources

Mary Thompson, William Johnston, Srilekha Mudumbai, Gary Hoo,
Keith Jackson, Abdelilah Essiari

Information and Computing Sciences Division
Ernest Orlando Lawrence Berkeley National Laboratory
Berkeley, CA, 94720
pkidev@george.lbl.gov

Abstract

We have implemented and deployed an access control mechanism that uses digitally-signed certificates to define and enforce an access policy for a set of distributed resources that have multiple, independent and geographically dispersed stakeholders. The stakeholders assert their access requirements in use-condition certificates and designate those trusted to attest to the corresponding user attributes. Users are identified by X.509 identity certificates. During a request to use a resource, a policy engine collects all the relevant certificates and decides if the user satisfies all the requirements. This paper describes the model, architecture and implementation of this system. It also includes some preliminary performance measurements and our plans for future development of the system.

1. Motivation: Distributed Computing Environments

In distributed computing environments such as research collaborations spanning several institutions, there may be independent and geographically dispersed individuals with authority to control access to the resources [18]. We wish to provide an automated system to allow these *stakeholders* to assert their authority over a resource in a flexible manner, consistent with the scope of their authority. Our immediate motivation is to enable sharing over open networks of valued resources within the scientific community generally, and for distributed collaboratories in the DOE2000 project [8] in particular.

The Department of Energy (DOE) supports a number of collaborative research environments in which people from universities and companies work with DOE national laboratory personnel and resources. The Lab resources consist of large scientific instruments such as electron microscopes or high-energy light sources; supercomputers or other high-end compute servers; and large-scale storage systems. Researchers from any of the groups may contribute software and data to be shared by other members of the group. The owners of software and data want to be able to securely store data or use their software on hardware that is owned by another entity. The owners of hardware resources want to be able to control their uses. These stakeholders may want to share with some, but not all members of the collaboratory. For example, commercial members may want to use common compute hardware, but share their data and results only with other members of their own organization.

Such an environment requires that the stakeholders be able to enforce access policy on their resources even when those resources are physically controlled by a different administrative domain. Each stakeholder must be willing to trust that the resource server will enforce access control, but the stakeholder should be able to flexibly specify access requirements for its resources.

Traditionally, stakeholders have relied on access control lists (ACLs), stored on the resource server, to express access policy. However, such ACLs typically require a central administrator to make all changes, which means both that the administrator must be trusted by all stakeholders and that the administrator is potentially a bottleneck to rapid updating of the policy. Also, ACLs usually require the server domain to maintain accounts and other administrative support for both stakeholders and users. These problems are all exacerbated when some or all stakeholders and users are administratively and geographically remote from the server.

Another problem that arises in distributed research environments is that there may be multiple principals from different administrative domains who need to have input to the access control policy for a single resource. The attempted execution of proprietary code (e.g., a large scientific modeling program) owned by a third party on a remote supercomputer is an example of such a multi-

This work is supported by the U. S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Mathematical, Information and Computation Sciences office (http://www.er.doe.gov/production/octr/mics), under contract DE-AC03-76SF00098 with the University of California. This document is report LBNL-42928

ply-controlled resource. The owner of the supercomputer may want to restrict the amount of run-time allotted to a user, and the author of the code may want to specify who may run the code. Getting permission to run the code, therefore, requires satisfying two separately administered policy requirements.

Multiple layers of management may wish to impose independent restrictions on the use of a large scientific instrument. For example, top-level administrators may have general restrictions based on nationality or membership in an organization, safety officers may require special training, and the principal investigator may have his own set of requirements for the project which has scheduled time on the instrument.

In these scenarios, the resource (data, instrument, computational or storage capacity) has multiple stakeholders and each stakeholder will impose conditions for access, called *use-conditions,* on the resource. All of the use-conditions must be met simultaneously in order to satisfy the requirements for access. Further, it is common that the principals in these scientific collaborations are geographically distributed and multi-organizational. Therefore, without reliable, widely deployed, and easily administered access control mechanisms it will not be feasible to administer a secure collaborative environment. The access control mechanism must allow secure, distributed management of policy-based access to resources and provide transparent access for authorized users and strong exclusion of unauthorized users, in an operating environment where stakeholders, users, and system/resource administrators may never meet face to face.

2. Goal: Policy-based Access Control

We want our access control mechanism to support, in a computer-based working environment, the same sort of distributed authority over resources that is used in non-computer group endeavors. Each stakeholder should be able to make its assertions (as it does now by signing a policy statement) without reference to a mediator, and especially without reference to a centralized system administrator who must act on its behalf. The mechanism must be dynamic and easily used by all concerned – stakeholders and users – while maintaining strong assurances. The solution should scale with the number of stakeholders, resources and users.

Specifically, the access control mechanism should be able to collect all of the relevant assertions (identity, stakeholder use-conditions, and corresponding user attributes) and make an unambiguous access decision requiring an absolute minimum of centrally administered configuration information. Once the policy-based decision is made, the resource server should be able to ensure compliance both on the part of the intended users and unauthorized parties. The mechanism should also be based on, and evolve with, emerging, commercially supplied, public-key certificate infrastructure components.

3. Approach: Certificate-based Distributed Security

Our approach to policy-based access control in a distributed environment is based on digitally signed documents, or *certificates,* that convey identity, authorization, and attributes. A digital signature can assert document validity without the physical presence of the signer or physical possession of documents signed in the author's handwriting. The result is that the digitally signed documents that provide the assertions of the stakeholders about a resource, or assertions of trusted authorities about attributes of a user, may be generated, represented, used, and verified independent of time or location.

Users are authenticated by presenting an X.509 identity certificate [17] and proving that they know the associated private key. These certificates are issued by *certificate authorities (*CA) that verify the connection between a person or system component and possession of a public key / private key pair. Stakeholders create and digitally sign *use-condition certificates* that define conditions that must be satisfied by a user before being given access to a resource. *User attributes* are asserted by "authorities" that provide assured information as digitally signed *attribute certificates* [11]. Both use-condition and attribute certificates may be stored local to the issuer as long as they can be provided by a server when they are needed to determine permissions during an access request.

Components that enable the use of these certificates include reliable mechanisms for generating, distributing, and verifying the digitally signed documents; mechanisms that match use-conditions and attributes to decide if access should be allowed; and access methods that enforce policy for the specific resource based on the access control decision. All of these mechanisms rely on public-key cryptography for digital signatures, a public-key infrastructure for certificate management and a protocol for secure, authenticated communication, such as the Secure Sockets Layer protocol (SSL) ([25], [26]). (For a general introduction to public-key technology see [12] or [24].)

A frequently asked question is how is the PKI-certificate approach is better than the well established DCE/Kerberos access control system. Kerberos and DCE allow remote users secure access to centralized resources.

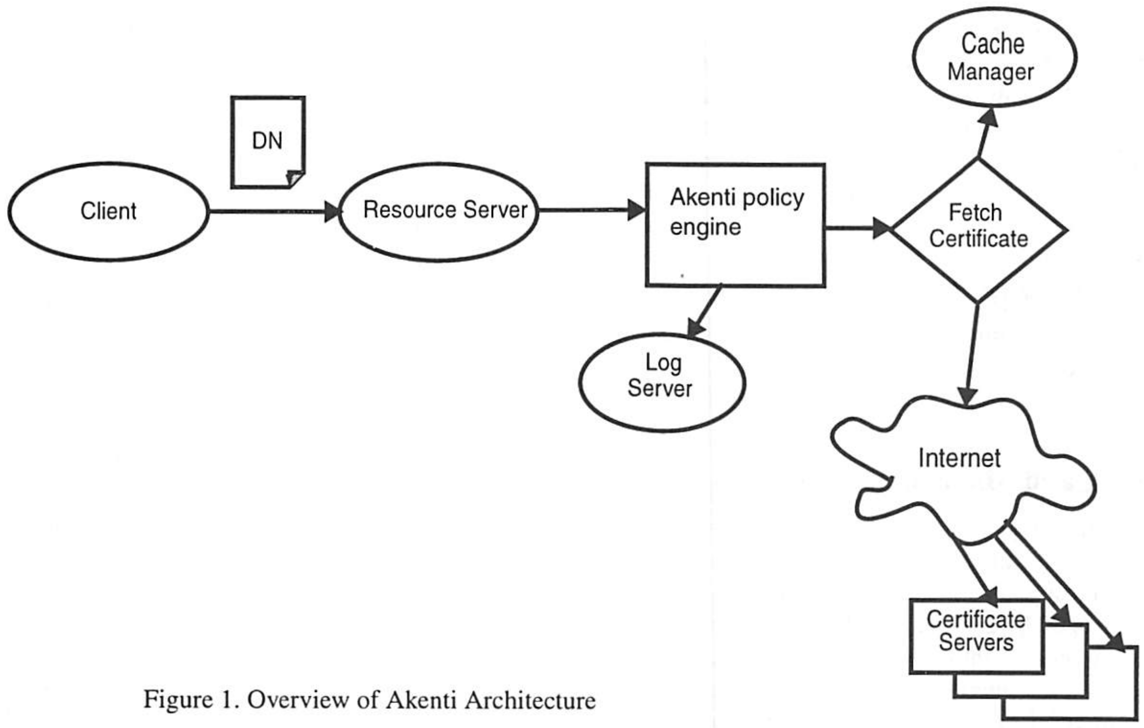
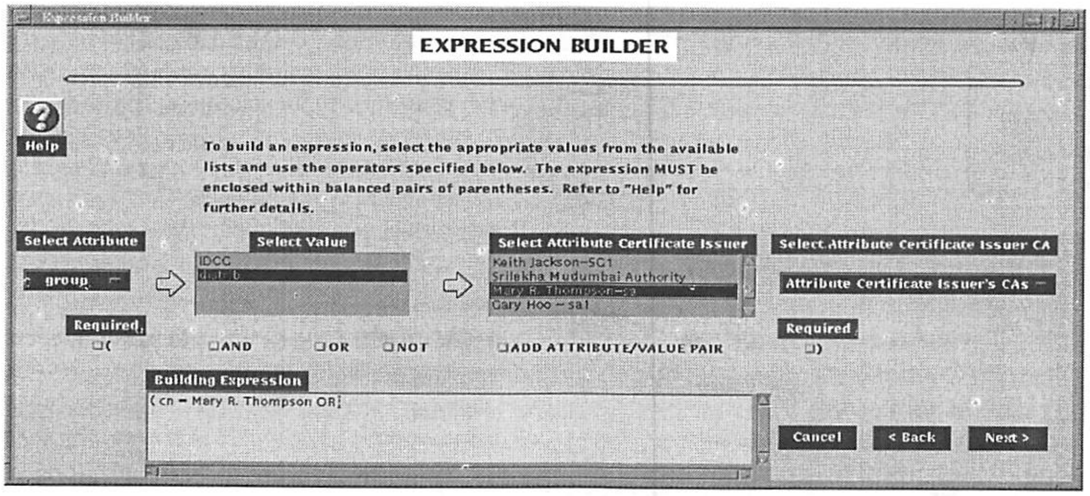


Figure 1. Overview of Akenti Architecture

Access by users from multiple administration domains can be addressed by establishing cross-realm trust. In the DCE environment the policy about a set of resources is defined by an ACL in the realm which controls the resource. Only user identities and group memberships are assigned by the other realms. Since the PKI approach allows pieces of the access control policy to be stored in distributed certificates, it enables distributed stakeholders to more easily control their resources. It is also our expectation that PKI identity certificates will become more widespread than Kerberos identities since many enterprises are beginning to use them for employee identification.

4. Architecture and Implementation

We are implementing a certificate-based access control system called Akenti [1], and initially deploying it in the DOE2000 Diesel Combustion Collaboratory [7]. Figure 1 shows the overview of the Akenti Architecture. The heart of this system is the *Akenti policy engine*, which gathers and verifies certificates and then evaluates the user's right to access to the requested resource based on these certificates. Figure 1 shows the major components of the run-time architecture. The *client* requests an operation on the resource and presents an identity certificate for authorization. The *resource server* authenticates the



To build an expression, select the appropriate values from the available lists and use the operators specified below. The expression MUST be enclosed within balanced pairs of parentheses. Refer to "Help" for further details.

(cn = Mary R. Thompson OR

Figure 2. A screen from the use-condition certificate generator

certificate and then asks the Akenti *policy engine* for an access decision. Akenti checks with a *cache server* for possibly cached certificates and if that fails, searches *certificate directories* across the internet. Akenti also logs all of its actions with a *log server.* Once Akenti has all the necessary certificates, it returns its access control decision to the resource server. The resource server then acts on that decision to perform or deny the operation on the resource on behalf of the client.

One other essential component of the Akenti infrastructure is the set of tools that generate certificates, query the policy and display the run-time operation of the system.

Generating and Managing Certificates

Akenti uses three types of persistent certificates: X.509 user identity certificates, use-condition certificates, and attribute certificates. The identity certificates are generated and managed by certificate authorities, such as the Netscape CA server, Entrust server or Verisign. These CAs provide a Web interface that allows the creation or revocation of certificates; a directory server (usually an LDAP server) to provide the certificates for use by applications; and Web browser to manage the certificates for the user. CAs typically support a *certificate revocation list* (CRL) mechanism that can be queried when an application needs to verify a certificate. Our implementation uses the Netscape CA, which was the easiest to install and run in a research environment. It currently only checks with the directory server to verify that a certificate has not been revoked. Some minor code addition would be necessary to use a CA's CRL mechanism, rather than just relying on the directory server to provide only non-revoked certificates.

The format of the attribute and use-condition certificates is defined by Akenti and consists of a list of ASCII keyword and value tuples which are signed by the issuer. These values include a validity period and a unique ID for the certificate. We have written two Java applications to help the user generate and sign these certificates. An example of the use-condition certificate generator's interface for specifying a use-condition is shown in Figure 2. These applications know how to query the resource server and CA directory server to provide a menu of reasonable choices for the stakeholder or attribute issuer to use in creating a certificate. The generators use a configuration file to find the resource server and the user's signing keys. The resulting certificates can be stored in directories chosen by the user that are accessible via a Web server, an LDAP server, a file system, or an on-line MSQL database.

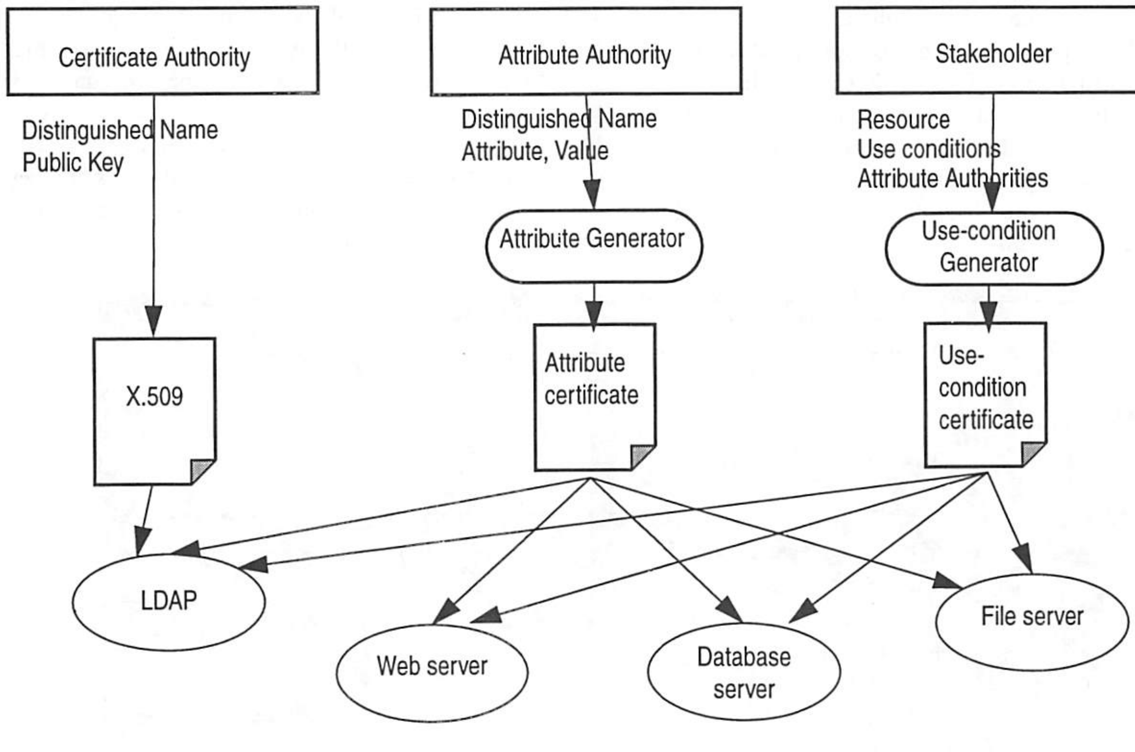


Figure 3. Certification Generation and Storage

```
-----BEGIN TEXT CERTIFICATE-----
-----BEGIN TEXT-----
use-condition
UID "portnoy.lbl.gov#1bea61fe#Mon Feb 01 00:17:11 PST 1999"
notValidBefore 981215014732Z
notValidAfter 991215014732Z
issuerAndCA "/C=US/O=Lawrence Berkeley National Laboratory/OU=ICSD/CN=IDCG-CA" "/
  C=US/O=Lawrence Berkeley National Laboratory/OU=ICSD/CN=Mary R. Thompson-sa"
resource http://imglib.lbl.gov/AkentiDist
scope sub-tree
attribute "( o : Diesel Combustion Collaboratory OR group : distrib )"
enable access read,execute
attributeIssuerAndCA o "Diesel Combustion Collaboratory" X509 "/C=US/O=Diesel Combustion
  Collaboratory/OU=SNL/CN=DieselCert.ca.sandia.gov" "/C=US/O=Diesel Combustion Collabo-
  ratory/OU=SNL/CN=DieselCert.ca.sandia.gov"
attributeIssuerAndCA group "distrib" Attribute "/C=US/O=Lawrence Berkeley National Labora-
  tory/OU=ICSD/CN=IDCG-CA" "/C=US/O=Lawrence Berkeley National Laboratory/OU=ICSD/
  CN=Mary R. Thompson-sa"
subjectCA "/C=US/O=Lawrence Berkeley National Laboratory/OU=ICSD/CN=IDCG-CA"
subjectCA "/C=US/O=Diesel Combustion Collaboratory/OU=SNL/CN=DieselCert.ca.sandia.gov"
-----END TEXT-----
-----BEGIN SIGNATURE-----
ZbO6puCmJGMY8Yz39RQ6Mf9Hx21+lC34suSH6onZ8Ml4CHVW+UHqQx6qShMe8D743+HR
  QPVDupsl
-----END SIGNATURE-----
-----END TEXT CERTIFICATE
```

Example 1. Use-condition Certificate

```
-----BEGIN TEXT ATTRIBUTE CERTIFICATE-----
-----BEGIN TEXT-----
attribute-certificate
attribute group
value distrib
notValidBefore 981215014732Z
notValidAfter 991215014732Z
subject "/C=US/O=Lawrence Berkeley National Laboratory/OU=ICSD/CN=Mary R. Thompson"
  "/C=US/O=Lawrence Berkeley National Laboratory/OU=ICSD/CN=IDCG-CA"
issuer "/C=US/O=Lawrence Berkeley National Laboratory/OU=ICSD/CN=Srilekha Mudumbai-
  sa" "/C=US/O=Lawrence Berkeley National Laboratory/OU=ICSD/CN=IDCG-CA"
-----END TEXT-----
-----BEGIN SIGNATURE-----
LedD9aawMkhpmW2dzt+o10Qb0Eanen0qMnYyAGYWPNL6DzbVqBIBXFesze40jPN6WelbV
  KL8SCP1Q/-----END SIGNATURE-----
-----END TEXT ATTRIBUTE CERTIFICATE-----
```

Example 2. Attribute Certificate

Figure 3 summarizes the different entities that create certificates, the tools that they use and the servers that manage the certificates. Not all types of certificate servers need to be available.

Let's look at an example of a use-condition and attribute certificate and what access they grant. The use-condition certificate in Example 1 is for the resource referenced by the name *"http://imglib.lbl.gov/AkentiDist"* and any directories under it. It was issued by the entity *Mary R. Thompson-sa*; it allows read and execute access to the resource. The user must either be a member of the organization *Diesel Combustion Collaboratory*, as attested to by presenting an identity certificate issued by the Diesel Lab CA, or be a member of the group *distrib,* as attested to by *Srilekha Mudumbai-sa*, whose signature must match the public one found in an ID certificate issued by the LBNL CA. The user must have an identity certificate issued by either the Diesel Lab CA or the LBNL CA.

The attribute certificate in Example 2 attests to *Mary R. Thompson* being a member of the group *distrib.* Since it has been issued and signed by *Srilekha Mudumbai-sa* it would satisfy the proceeding use-condition certificate.

Resource Server

Our model includes a resource server that interfaces to resources on behalf of the client. It is responsible for establishing a secure and authenticated connection with the user. Our current applications use an SSL-enabled Apache Web server ([2], [3]) and an Orbix SSL-enabled Object Request Broker (ORB) [22]. The SSL protocol is configured to require client side authorization. In this mode mutual authentication is performed and at the end of the handshake the server has an authenticated X.509 identity certificate for the user, which can be used to securely grant authorization to the entity at the other end of the encrypted connection.

After authentication, the server calls the Akenti policy engine with the DN of the user and the name of the resource that is being requested. The policy engine either returns "access denied" or a list of actions that are allowed. The resource server must know how to interpret and perform the named actions on behalf of the client.

Akenti Policy Engine

The policy engine is a library module that finds all the use-condition certificates that apply to a resource. It verifies that each use-condition certificate has been signed by a legitimate resource stakeholder and has not expired. Then for each use-condition that must be satisfied, the policy engine searches for the attribute certificates that attest to the required values for the user. Once all the use-conditions and attribute certificates are gathered and verified, the policy engine evaluates what actions, if any, the user is allowed to perform.

Since one of our goals is to avoid centralized access policy information, we immediately face the problem of where to look for use-condition certificates and how to know that they have all been found. Our solution introduces a minimal *authority file* that is stored with the resources. This file contains a list of servers which supply identity and attribute certificates; the list of the use-condition issuers (stakeholders); and where the use-condition certificates are stored. In addition, there is a *root authority file* that contains the list of trusted CAs, and their public keys, for the whole resource tree.

The Akenti policy engine searches each of the use-condition certificate directories listed in the authority file. It must find at least one use-condition certificate for each stakeholder. If a stakeholder supplies no use-condition certificate, Akenti denies all access to the resource. Several assumptions underlie this behavior. First, a stakeholder's use-condition certificates for a resource must all be stored in one place, so that if one is found, they all are. Second, each stakeholder intends to place a use-condition on the resource. (It is also possible to specify joint stakeholders, where a use-condition from either one will suffice.) Third, it is better to deny all access to a resource if the input from one of the stakeholders is missing, than to erroneously grant access that the missing use-condition would have denied. Finally, that the stakeholders will store their use-condition certificates in a secure and reliably accessible place. One option is for the resource server to provide a secure LDAP server on which the stakeholders may store their certificates.

Naming the resources is another issue that needs to be addressed. Our model assumes that the resources may form a hierarchy, such as a file system or a tree of Web documents. This model can obviously be reduced to a single level for something like a scientific instrument. Grouping resources into a hierarchy that reflects the desired protection reduces the number of use-condition certificates that must be issued. A use-condition has a scope of either local or sub-tree: Example 1, for instance, shows a sub-tree-scoped use-condition. A locally scoped use-condition only applies to the level named in the use-condition certificate; a subtree-scoped use-condition applies at that level and at any level beneath it. The name of the resource in the use-condition certificate is typically the name used to reference the resource. Hence, URLs are used for Web-accessed resources, CORBA object names can be used in the context of an ORB, etc.

There is one more non-obvious feature of use-conditions. Some use-conditions grant general access to a resource, as specified by "*enable access*" either as the only access or in conjunction with actions such as read or write. If a use-condition specifies access, a user must satisfy it before gaining any access to the resource. This feature allows stakeholders to exercise veto power over any subtree of resources. In particular, we envision this feature being used at the top level of a resource hierarchy, where a global use-condition might require that any user of the resource meet a condition specified by a high level authority, e.g., that all users must be member of some organization or group of organizations.

If a use-condition only grants actions, then any user who satisfies it is granted those actions in addition to whatever other actions she may be allowed. One use for such a certificate would be to grant "write" or "modify" privileges to a small subset of people while a larger group would be granted "read" access. For example there could be three use-conditions that apply to a resource: a subtree-scoped one at the root level that only grants access to everyone in the organization

LBNL; a second one at a local level that grants read permission to everyone in a group "readers", and a third one at the local level that grants "modify" permissions to a group "writers". Thus the user's identity certificate would need to show her belonging to the organization LBNL, and she would need an attribute certificate placing her in either the "readers" or "writers" group before she would be allowed any access to the resource. If she also has a certificate placing her in the other group, she would get the additional access. Note that we use identity certificates in place of a separate attribute certificate to attest to the user's values for selected components of a DN: organization, organizational unit and common name.

All use-conditions must be evaluated before a user's access can be determined. All those that grant "access: must be satisfied; any that assert negative conditions, e.g., not from a proscribed country will take precedent and deny access; any of the positive ones may add more rights. It is certainly possible for multiple stakeholders to impose contradictory use-conditions, which may result in no access to the resource being granted. We believe that this mirrors the way stakeholders wish to impose control. The solution is for the stakeholders to be able to easily see what the combined results of all the use-conditions is, and to co-operate with the other stakeholders to create a set of use-conditions that satisfies everyone.

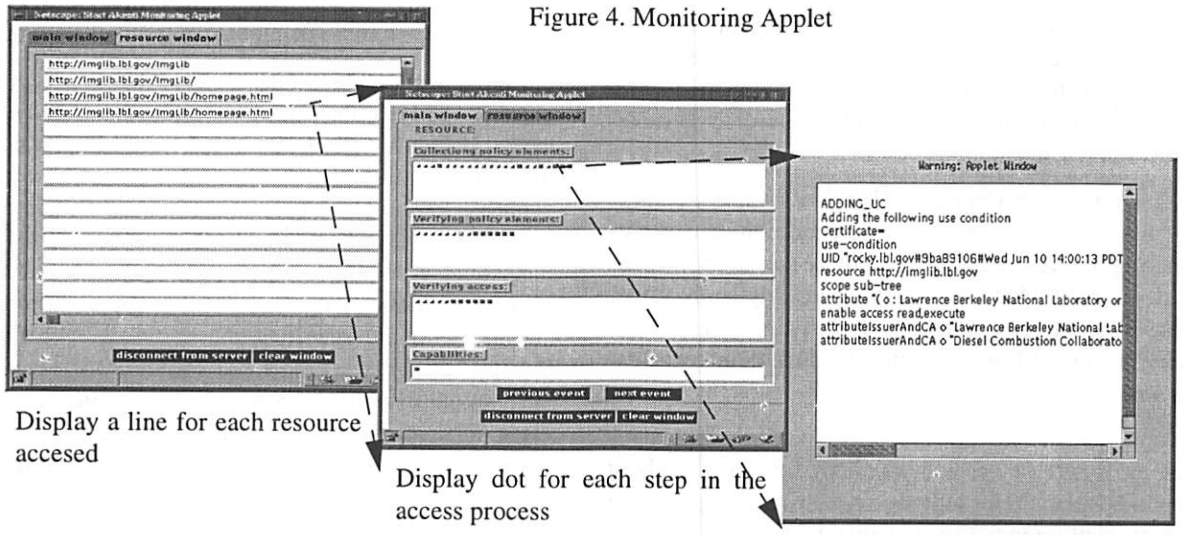
5. Implementation Refinements

Caching

Since Akenti's access control decisions are all based on the contents of signed certificates that are distributed across the network, an obvious performance bottleneck is the gathering of these certificates. One way to improve this performance is to make the searching as efficient as possible. We have done this by carefully choosing the parameters to the LDAP and database searches so as to maximize the amount of filtering on the server side and reduce the number of non-applicable certificates that are returned to Akenti. Certificates that are stored in file systems or Web directories are named by a hash index of the search attributes. Akenti's search is based on these hash names.

The other obvious way to minimize searching over the network is to cache certificates locally once they have been found and verified. Since the Akenti policy engine is a library module without any persistent memory, we have implemented a cache server to store the certificates. In the case of the Web server used as an access control gateway, several processes may be checking access for the same set of resources in parallel and all talking to the same cache server. In our current implementation, the cache server runs on the same machine as the resource server, but it could easily run on a separate machine if desired. The cache manager caches use-condition certificates, identity certificates, and attribute certificates.

In addition to static certificates, Akenti creates, signs and caches a certificate containing the access rights of a user for a resource. This certificate is in effect a dynamic *capability certificate*. Capability certificates are especially useful for a hierarchical collection of Web docu-


Netscape: Start Akenti Monitoring Applet
main window
resource window
disconnect from server
clear window
RESOURCE:
Collecting policy elements:
Verifying policy elements:
Verifying access:
Capabilities:
previous event
next event
Warning: Applet Window
Display a line for each resource accesed
Display dot for each step in the access process
Display log message for each dot.

```
http://imglib.lbl.gov/ImgLib
http://imglib.lbl.gov/ImgLib/
http://imglib.lbl.gov/ImgLib/homepage.html
http://imglib.lbl.gov/ImgLib/homepage.html
```

```
ADDING_UC
Adding the following use condition
Certificate=
use-condition
UID "rocky.lbl.gov#9ba89106#Wed Jun 10 14:00:13 PDT
resource http://imglib.lbl.gov
scope sub-tree
attribute "( o : Lawrence Berkeley National Laboratory or
enable access read,execute
attributeIssuerAndCA o "Lawrence Berkeley National Lab
attributeIssuerAndCA o "Diesel Combustion Collaborato
```

Figure 4. Monitoring Applet

ments. A Web browser often makes several independent access requests for each logical document request, as when a page includes images. Also, in hierarchical collections of Web documents, related documents frequently all inherit access permissions from the same directory. Once the capability is cached, subsequent references short-circuit most of the Akenti policy engine.

The validity period for a cached certificate needs to be under the control of the stakeholders of the resource for which the certificate is going to be used. Since the same identity or attribute certificates may be used for several resources, checking the validity of a certificate requires two steps. When the certificate is first cached, the "not-valid-before time" is set to the current time, and the "not-valid-after time" is set to current time plus the validity period corresponding to the current resource. This value can be found in the authority file for the resource or can be the default value for the resource tree. If the cached certificate is going to be used to allow access to a different resource, the "not-valid-before time" stored with the cached certificate plus the validity period for the desired resource must be less than the present time. We make one exception to the above rule: since capability certificates represent the sum total of many different certificates, their lifetimes are kept very short (currently 5 minutes).

Monitoring

In order to provide a meaningful service, access control must be applied in such a way that the resources are protected as intended by the stakeholders. This involves understanding the structure of the resources and how they should be used, and developing a policy model that will support the intended access control. Akenti provides several services which are intended to help the stakeholder understand the policy implemented for the resource. In our prototype these services are available to anyone who has access to the resource tree. In a more restrictive environment, they could be limited to stakeholders of the specified resource.

First, a remote user can ask Akenti to statically display all the authority files and use-condition certificates applying to a resource. A stakeholder can use this facility to discover what use-conditions already apply, either those inherited from a higher level in the resource hierarchy or imposed by other stakeholders, before designing a new one.

Second, the Akenti policy engine performs extensive logging in order to provide dynamic information about the its behavior. The logging uses an existing logging library, NetLogger [27], that directs the logging messages to a log server and/or file or standard output depending on how the resource server has been configured. Directing the messages to a log server allows the information to be presented in real time to a user who is attempting to gain access. We have written a Java applet that graphically displays each step of the access-granting process. (See Figure 4.) The log is also saved to an audit file on the resource server so that the administrators or stakeholders may monitor the use of their resources.

6. Status

Over the past two years, we have implemented several Akenti-enabled servers. The prototype that has seen the most use is an Apache Web server with the SSLeay patches [3] and with Akenti replacing the standard access control module. It is being used as an access control gateway for a variety of Web-based resources within the Diesel Combustion Collaboratory [7]. This is a prototype collaboratory that involves two independent CAs and a number of government and commercial organizations scattered around the country. A single Akenti/Apache Web server is used to control access to two different image and data archives and a Web-based electronic notebook developed by Oak Ridge National Laboratory [14]. In these applications, the Akenti policy engine is called both by the Apache Web server and then again by the scripts that are used for fine-grained control of the resources. The Akenti policy engine is wrapped in a main program that is executed by scripts. Once the user has an identity certificate and the correct credentials, the access control is almost transparent.

We have also implemented a prototype CORBA ORB. Minor changes were made to the client side to find and present an identity certificate and to the server side to use one of the Object Management Group (OMG)-defined interceptors to call the Akenti policy engine.

Although we are just starting to evaluate Akenti, it appears to provide the sort of distributed management of access policy by multiple stakeholders that was our goal, while enforcing strong access control over the resources. In the case of remote references to resources, the additional overhead of Akenti does not seem to be unreasonable. We have invested considerable effort in creating user-friendly interfaces for the stakeholders. Currently we are rewriting the certificate generators. This is due partly to the rapid evolution of Java, and partly to our discovery that as we create policies for different sorts of resources and user populations, we find we need more flexibility and clarity in the certificate generator interfaces.

When problems occur, the logging facility is now both complete enough and sufficiently accessible that a practiced user can figure out what credential is missing, expired, etc. Probably the biggest problem with the logging information is that there can be too much of it, and sorting out the real cause of the problem is sometimes difficult. This is an area of ongoing development.

7. Vulnerabilities

The major vulnerability of the system derives from the fact that while stakeholders and their repositories are named in the authority file on the server, the use-condition and attribute certificates they depend on are maintained on distributed, "trusted" servers. If those certificate servers are not secure, then certificates could be deleted, resulting in an unintended access control policy. The type of failure depends on the type of certificates that are missing. If none of a stakeholder's use-condition certificates is available there will be a complete denial of access. If only some of a stakeholder's use-condition certificates are missing, the access could be greater than it should be. And if an attribute certificate is missing, specific users may get more or less access than they should.

The problem where a missing use-condition certificate allows greater permission than desired, can be solved by requiring the stakeholder to put all the use-conditions into one certificate. If that certificate is missing all access is denied. This constraint will produce more complicated use-condition certificates which may make the policy harder to understand.

The fact that a missing attribute certificate could permit too much access was revealed when comparing Akenti to KeyNote (see Section 9). The current use-conditions allow negative constraints, e.g., not belonging to some proscribed group. If that group certificate is missing, the user may get access to a resource that should be denied. To prevent this, use-conditions must always phrased in terms of positive conditions, so that attribute certificates will always increase access.

As we gain more operational experience, we will be better able to assess the importance of each of these vulnerabilities and the trade-offs required to address them.

8. Performance Measurements

The performance of a system composed of a client and a remote server using Akenti access control is often dominated by factors that are not controllable by the system designer. Nevertheless, it is valuable to measure the system performance in order to characterize the variables and to optimize those that can be influenced by the system.

Two variabilities arise from the fact that this is a distributed system: the network transmission time between the client and server, and the network transmission time plus the server response time between Akenti and the certificate servers. Performance factors under loose control by the stakeholder are the number of certificates required to make an access decision and the volume of data that is passed between the client and server. Finally, there is the overhead directly attributable to Akenti which includes the time associated with establishing an SSL connection and encrypting the data between the client and server, and the time spent in the Akenti policy engine gathering and verifying certificates.

The measurements in this paper are for file fetches between a Java SSL-enabled client an Akenti/Apache Web server. The client, server and all the certificate servers are on a 100 Mb/s LAN. The document sizes varied between 1KB and 1MB in order to evaluate the overhead of the SSL encryption. The SSL keys are 128-bit keys. The number of certificates that were required to permit access varied between a minimum of one use-

	No caching			Caching		
	Akenti (seconds)	**SSL/network (seconds)**	**Total (seconds)**	**Akenti (seconds)**	**SSL/ network (seconds)**	**Total (seconds)**
Min-acc						
1K	**0.86**	**0.65**	**1.51**	**0.20**	**0.65**	**0.85**
1M	**0.90**	**1.75**	**2.65**	**0.22**	**2.02**	**2.34**
Ave-acc						
1K	**2.26**	**0.73**	**2.96**	**0.115**	**0.646**	**0.762**
1M	**2.24**	**1.96**	**4.00**	**.188**	**1.77**	**1.96**

Table 1: Average times to fetch a document from a Secure Akenti server.

condition certificate and two identity certificates and a more average case that required two use-condition certificates, one attribute certificate, and four identity certificates. We also took measurements with and without Akenti server caching enabled.

On the server side, Akenti does extensive logging of each logical step in the policy engine. This measurement excludes the server side time spent in the Apache server and SSL encryption. The times in Table 1 are the times in the Akenti policy engine (Akenti), the total socket read time the client saw (Total) and the difference between the two which can mostly be accounted for by SSL overhead (SSL/network).

The test program fetched the same file 10 times and then calculated the mean fetch time. The data from the client program was combined with the matching server log entries to determine the values in Table 1. Each case was run several times in succession to average over variations in the network load. Thus each number is the average of about 30-40 file fetches.

Several observations can be made from this data. As the files get bigger, the SSL encryption times tend to dominate the overhead. However, SSL can be configured to do only authentication and message integrity checking if encryption is not needed, which would reduce this time. As more certificates are required to grant access, the times in the Akenti policy engine increase. We can see from the Akenti log files that the major categories of time in the policy engine are fetching certificates and verifying signatures. In the minimum certificate case about 79% of the total time was spent fetching certificates and 9% was spent on signature verification. In the average certificate case, about 83% of the time was in fetching certificates and 8% was spent in certificate verification. Failing to find a certificate, such as an optional attribute, took more than twice the time of successfully finding one. The rest of the time was split between reading authority files, parsing the use-conditions and general program overhead.

In the case when caching is enabled and a valid capability certificate is found, the time in the policy engine is about 0.11 seconds. The variation that appears in these times in Table 1 is the result of the capability timing out and having to be reestablished. The caching lifetimes of cached use-condition and identity certificates is generally longer than that for capabilities, so cached versions of those certificates may be used when reestablishing the capability.

For a Web server that is mainly fetching documents, caching by the Akenti policy engine provides a big performance benefit, since there are usually several clustered access to documents in the same general protection domain. If Akenti is being used by a server where the pattern of accesses is isolated, the caching may actually be a disadvantage, since cache misses and subsequent cache updates are relatively costly.

	With Akenti (seconds)	**No Akenti (seconds)**
1Kbyte	**0.76**	**0.02**
1Mbyte	**1.96**	**0.75**

Table 2: Document fetch with and without Akenti access control

The corresponding time that it took to fetch 1KB and 1MB files using the same client program but a standard Apache web server with no access control was about 0.02 seconds for the 1KB file and between 0.69 and 0.80 seconds for the 1MB file. (See Table 2.) For Web servers, that most meaningfully compares to the 0.76 and 1.96 second times for an average set of access constraints from a caching Akenti server. Obviously, the target applications for Akenti access control are ones where there is something important to protect and the granularity of the access checking is fairly large, e.g., a large document to be fetched, or a substantial process is to be started on the resource machine.

Another case where the Akenti overhead is not too severe is accessing a Web document that requires the parallel fetching of many secure components. For example, a document where all its parts are in the same protected tree. In this case the browser and the server fetch in parallel, and since Akenti has no trouble working in parallel and sharing the same cache, the net result of such a clustered fetch is not too much worse than the secure fetching of one document. For example, for a document containing 10 images, the html frame is retrieved in 4 seconds, the first three images appear after 8 seconds and the rest of the images appear at 10 seconds. The corresponding behavior for such a page with no access control is for all the images to appear after about 1 second.

9. Related Work

Currently there are several other projects working with signed certificates to enable access control decisions. One of the earliest attempts to define standards was the Simple Public Key Infrastructure (SPKI) [9] IETF draft proposal by Ellison, et al. This work proposed a standard for authorization certificates, name certificates and access control lists that are all represented by a formalized key-word, value syntax called S-expressions. An authorization certificate consists of an issuer, represented by a public key; a subject also represented by a public key; an optional delegation field; the actions that are authorized; and an optional set of real-time constraints on the certificate. We chose not to base Akenti on authorization certificates, but on policy and use-condition certificates instead. Our decision sidesteps the problem of revoking authorization certificates, and makes the policy more explicitly stated and thus easier for multiple stakeholders to understand. Our design also only allows only one level of delegation, rather than the possibility of delegation on each certificate. Again this makes the policy more restrictive, but easier to discover.

Pekka Nekander and Jonna Partanen of Helsinki University of Technology have used the SPKI-style certificates to carry access permissions with Java code, rather than relying on local access control configuration files [21]. They argue that maintaining a local security configuration on each machine on which a Java class might be executed does not scale to large numbers of machines and classes. This argument is parallel to our observation that requiring all users to have accounts and to be entered into ACLs on each resource server does not scale to separately administered distributed compute resources.

Nekander's and Partanen's system uses authorization certificates to grant a Java class (or JAR file) specific permissions. If the machine on which the class is to be executed accepts the signer of the certificate, the class is allowed to perform the actions. This type of access decision supports the Java security model which grants access to code on the basis of where it came from rather than who caused it to be executed. However, the focus in Akenti is on allowing access to resources, which are referenced through a resource server, by specified classes of users. Thus it is more natural to base access decisions on authenticated identity certificates matched against policy requirements at the time of resource use.

The PolicyMaker [5] and KeyNote [4] trust management systems present a very generalized approach to specifying and interpreting security policies, credentials and trust relationships. Both of these systems share with Akenti the goal of putting all the policy and credential information into signed certificates that can be stored in a distributed fashion. These certificates are then gathered together by a policy engine module or daemon at the time of resource access and interpreted to allow or deny the access. PolicyMaker certificates are much more generalized than Akenti's as they consist of programs written in a general programming language. Since an access policy is represented by the set of all such certificates, it may be very hard to understand the overall access policy for a resource. The KeyNote system settled on a C-like expression and regular expression syntax for describing conditions, similar to that which Akenti uses. However, in an Akenti application the policy for a resource use is spelled out in a few authority files and use-condition certificates which make explicit what actions users or classes of user have while in an authorization credential system such as KeyNote, all the assertions combine to imply a policy.

KeyNote certificates also differ from Akenti's in the principle of "assertion monotonicity" which means that each assertion will increase the permitted actions. Akenti permits a use-condition to specify negatives, e.g., that a person may not be a member of a given group. Then if the corresponding certificate that identifies the person as a member of the proscribed group is not found, the person may be granted more permissions than were intended by the stakeholder. Future versions of Akenti should close this loophole.

A more specific use of identity certificates for authorization can be found in the Globus Project [15], a joint research project between Argonne National Laboratory and the University of Southern California's Information Sciences Institute (USC/ISI). Globus is developing a software infrastructure for distributed computational and informational resources, and is experimenting with using X.509 identity certificates to provide a one-time per session sign-on to a distributed set of resources. [13]

The Globus infrastructure provides a gateway server at each site. The server on the client side accepts and verifies a user's identity certificate and creates a temporary proxy certificate to represent the user to other Globus servers. The server at the resource site authenticates the certificate that it receives and maps the identity into a local user ID. The resident operating system then performs access control as usual. This solution was motivated by the early use of Globus to provide remote access to supercomputers. The administrators of the supercomputing sites were not interested in relying on a new access control mechanism. The Globus solution allows a user to authenticate once per session by presenting an identity certificate. However, each user must still have a local user ID and account with each resource server. We are working with the Globus group to integrate Akenti as an alternative access control mechanism for those sites that want distributed policy management.

10. Future Work

There are two general directions for future work on Akenti. One is to implement the policy engine as a daemon in addition to the current library module approach. The second is to improve the syntax of the various certificates. Our intent is to define an XML-based (Extensible Markup Language) [10] format for our certificates. XML has the advantages of presenting self-describing documents and being widely used by various scientific disciplines. There are tools available for validating an XML document against its document type definition (DTD) which may be useful to the interface programs that are used to create the certificates. Our goal is to use a standard syntax which may be familiar to the people who have to write and understand the policy.

There are several applications which we intend to integrate with Akenti. We plan to use Akenti to make access decisions for a network-bandwidth quality of service (QoS) framework that is being developed at LBNL [16]. In order to conform to the standards of the QoS community, a policy server should communicate via the COPS (Common Open Policy Service) [6] protocol. This is one of the motivations for implementing Akenti as a server that will respond to requests for access control decisions. COPS is a statefull protocol that, among other things, allows the resource server to upload or download policy documents. This feature gives us the option of keeping the authority files with the resource tree and uploading them to the Akenti server, allowing it to be a stateless decision maker.

We are currently integrating Akenti security with a secure mobile agent system that is being developed at LBNL [19]. This system customizes the Java security model to enforce policy-based access control on mobile Java agents.

Another possibility is to implement the proposed standard Generic Authorization and Access control API [23] with Akenti. To do so would require adding some additional library interfaces.

Both COPS and GAA expect the policy module to be able to make decisions based on the time of day, the location of the requestor and, in the case of COPS, on some level of allowed quotas for the resource. In our current model, Akenti simply copies strings that represent actions from the use-condition certificates and returns them to the resource server which interprets them. The concepts of a time during which a user may use the resource, an allowed IP address or domain from which the request may come, and a quota for use of resources must be added to the policy engine. To accomplish this, Akenti will need to define a convention for naming the use-conditions for time, location and quotas and then checking for compliance. The last case requires getting a value for the amount of resources used from the resource server and storing it as an auxiliary certificate.

11. Conclusions

In a larger view, useful security is very much a risk management, deployment and user ergonomics issue. Once it has been determined what level of security is required, the hard problem is integrating that level of security into the end-user (e.g., scientific) environment so that it will be used, trusted to provide the protection that it claims, easily administered, and genuinely useful in the sense of providing new functionality that supports distributed organizations and operation. As large enterprises establish public key infrastructures and people become accustomed to using an identity certificate to authenticate themselves rather than a multitude of passwords, certificate-based access control will seem natural and be more easily understood.

Akenti facilitates the decentralized creation and management of policy certificates and the use of these certificates to make secure policy-based access control decisions. Distributed certificates permit the decentralized administration of shared resources which is an important goal in a collaborative research environment. We believe that our prototype Akenti implementation has demonstrated the viability of such a distributed system.

Acknowledgments

Case Larsen wrote many of the certificate-handling and cryptographic libraries used in the Akenti implementation. Bob Aiken and Mary Anne Scott of DOE/ ER/ MICS have been consistent supporters of this approach to security and access control.

Availability

An early version of the Akenti policy engine and certificate generators can be downloaded from the Akenti web site. (http://www-itg.lbl.gov/Akenti/)

References

[1] "The Akenti Approach", http://www-itg.lbl.gov/Akenti/

[2] "About the Apache HTTP Server Project", http://www.apache.org/

[3] "Apache-SSL", http://www.apache-ssl.org/

[4] M. Blaze, J. Feigenbaum, A.D. Keromytis, "The KeyNote Trust Management System", work in progress Internet Draft, March1999

[5] M. Blaze, J. Feigenbaum, J. Lacey, "Decentralized Trust Management System", *Proceedings of the 17th Symposium of Security and Privacy,* pp. 164-

175 IEEE Computer Science Press, Los Alamitos, 1996

[6] J. Boyle, R. Cohen, D. Durham, S. Herzog, R. Rajan, A. Sastry, "The COPS (Common Open Policy Service) Protocol", Internet Engineering Task Force Draft, work in progress

[7] "Diesel Combustion Collaboratory Homepage", http://www-collab.ca.sandia.gov/

[8] "DOE2000 Homepage", http://www.mcs.anl.gov/DOE2000/

[9] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Yloenen, "Simple Public Key Certificate", Internet Engineering Task Force Draft, work in progress

[10] "Extensible Markup Language" http://www.w3.org/XML/

[11] S. Farrell, R. Housley, "An Internet AttributeCertificate Profile for Authorization", Internet Engineering Task Force Draft, work in progress

[12] W. Ford, *Computer Communications Security: Principles, Standards, Protocols, and Techniques*, Prentice-Hall, Englewood Cliffs, New Jersey, 07632, 1995

[13] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, "A Security Architecture for Computation Grids", Proc. *5th ACM Conference on Computer and Communications Security Conference*, pg. 83-92, 1998.

[14] A. Geist, N. Nachtigal, "ORNL Electronic Notebook Project", http://www.epm.ornl.gov/~geist/java/applets/enote/

[15] "The Globus Project", http://www-fp.globus.org/

[16] G. Hoo, W. Johnston, "QoS as Middleware: Bandwidth Brokering System Design", submitted to the *High Performance and Distributed Computing conference* 1999.

[17] R. Housely, W. Ford, W. Polk, D. Solo, "Internet X.509 Public Key Infrastructure", Internet Engineering Task Force Draft, PKIX Working group, work in progress. Also see Ford & Baum, *Secure Electronic Commerce*, Prentice-Hall, pp 214-230

[18] W. Johnston, S. Mudumbai, M. Thompson, "Authorization and Attribute Certificates for Widely Distributed Access Control", *IEEE 7th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*- WETICE '98

[19] S. Mudumbai, A. Essiari, W. Johnston, "Anchor - A Secure Mobile Agent Toolkit", - submitted to Mobile Agents '99

[20] B.C. Neuman and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks", IEEE Communications Magazine, v.32, n.9, Sep 1994, pp. 33-38

[21] P. Nikander, J. Partanen, "Distributed Policy Management for JDK 1.2", *Proceedings of Network and Distributed System Security Symposium*, San Diego, CA, Feb 3-5, 1999

[22] OMG-CORBA homepage" http://www.omg.org

[23] T. Ryutov, C. Neuman, "Access Control Framework for Distributed Applications", Internet Engineering Task Force Draft, work in progress

[24] B. Schneier, *Applied Cryptography*, Second Edition, John Wiley & Sons, 1996

[25] "The SSL Protocol", http://home.netscape.com/eng/security/SSL_2.html

[26] "SSLeay FAQ", http://www.psy.uq.oz.au/~ftp/Crypto/

[27] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, D. Gunter, "The NetLogger Methodology for High Performance Distributed Systems Performance Analysis", *Proceedings of the IEEE High Performance Distributed Computing -7*, '98

Digital-Ticket-Controlled Digital Ticket Circulation

Ko Fujimura, Hiroshi Kuno, Masayuki Terada,
Kazuo Matsuyama, Yasunao Mizuno, and Jun Sekine

NTT Information Sharing Platform Laboratories
{fujimura, kuno, terada, matsuyama, mizuno, sekine}@isl.ntt.co.jp

Abstract

This paper presents a new digital-ticket circulating scheme and trust management scheme for a digital ticket. A digital ticket is a digital medium that guarantees certain rights of the owner and it includes software licenses, resource access tickets, event tickets, and plane tickets.

The circulation of digital tickets comprises three types of principal transactions: issuance, transfer, and redemption. Depending on the application, various conditions must be satisfied to execute these transactions, e.g., only qualified shops can issue the tickets and only a certain agent can transfer the tickets. This paper introduces circulation control tickets, which are required to issue, transfer, redeem a ticket, and proposes specifying the required control ticket types in the ticket to be circulated itself using the Generalized Ticket Definition Language. The ticket circulating system issues, transfers, or redeems a ticket only if the control tickets are owned by the participants of the transaction. The circulation control tickets themselves can be any type of digital ticket, e.g., a driver's license or a membership certificate to certain group, and these tickets can be recursively circulated in the ticket circulating system. This scheme provides the ticket circulating system with both the flexibility needed to match the business scheme of interest and application independence.

This paper also proposes a ticket-type-based trust management scheme that enables users to mechanically verify the trust of a ticket by the presented ticket type verification procedure.

1. Introduction

A digital ticket is a digital medium that guarantees certain rights of the owner and it includes software licenses, resource access tickets, event tickets, and plane tickets. A digital ticket covers a wide range of digital rights from a digital certificate [8][20], in which transferability is not required and where there are no restrictions on the number of times it can be consumed, to digital cash [2][14], in which transferability is required and restrictions on consumption apply.

We are developing a system that can circulate all tickets with various rights in a common manner. This system enables service providers to reduce the development costs of the ticketing software/hardware and also enables users to view and manage various tickets using a common "ticket wallet", which greatly improves usability.

The trust management scheme [1][3][6][7] developed for digital certificates and the double-spending protection scheme [2][14] developed for digital cash can also be applied to digital tickets as base technologies, since digital tickets have aspects of both digital certificates and digital cash. These technologies, however, do not provide solutions for the specific requirements of digital tickets, i.e., diversity in circulating requirements and business schemes.

To circulate various types of digital tickets using a common ticket processing system, we proposed a general-purpose digital ticket framework, in which a ticket is circulated by interpreting the ticket properties of anonymity, transferability, and divisibility, specified in the ticket itself using the Generalized Ticket Definition Language [9]. No circulation control scheme or trust management scheme for digital tickets have, however, been presented up to now. These issues are especially important since the requirements depend on the ticket's business scheme, and this makes generalization difficult. This paper focuses on these issues and presents a new approach that we implemented in a prototype system.

The following were taken as our design goals:

No centralized organization. Designs that rely on deference to a global, centralized organization should be avoided since a single failure in the organization may impair the entire system. No centralized broker who sells all types of tickets, or centralized authority that authenticates all issuers or other participants, should be assumed.

Business scheme flexibility. Unlike digital cash, various requirements must be satisfied when a ticket is circulated depending on the application. Examples include "only qualified agents can transfer the tickets", or "only a certain member of a group can redeem the tickets". To satisfy these business requirements, flexible control of ticket circulation is required.

Management autonomy. Responsibility for the ticket and for settlement when a task or service is not satisfactorily rendered should be application dependent. These management policies should be defined freely by the ticket issuer. For example, a ticket issuer should ask a certificate authority (CA) to endorse the contents of their tickets only if the issuer desires it.

Trust manageability. Diverse types of digital tickets will be circulated in the future. This makes it difficult for users to judge whether a ticket can be trusted or not. To support such judgement, a trust management system that mechanically verifies the trust of a ticket is thus required.

Simplicity. Simplicity is important in understanding the system, which is necessary for people to trust the system. It also minimizes the probability of a security hole resulting from an implementation error.

Prevention of duplicated redemption is also an important issue of the digital-ticket circulation system. This exceeds the scope of this paper since several double-spending protection schemes invented for digital cash can be applied to digital tickets. The digital ticket storage system, i.e., how digital tickets are stored in smart cards, PCs, or network, is also a major issue when implementing a digital-ticket circulation system. This point is also beyond the scope of this paper. We will address this issue in other papers.

To achieve these design goals, this paper introduces circulation control tickets, which are required for issuing, transferring, and redeeming a ticket, and specifies, using the Generalized Ticket Definition Language, the required control ticket types in the ticket to be circulated. The ticket circulating system issues, transfers, or redeems a ticket only if the control tickets are owned by the participants in the transaction. The circulation control tickets themselves can be any type of digital ticket, e.g., a driver's license or certain membership certificate, or other certificates issued by a certificate authority (CA), and these tickets can be recursively circulated using the ticket circulating system. This scheme provides flexibility to the ticket circulation schemes since various conditions can be defined. This scheme also provides management autonomy since any type of certificate can be used for endorsing the tickets without introducing complexity or a centralized organization.

This paper also proposes a new trust management scheme based on ticket type; the scheme defines the format and restrictions placed on ticket properties. In this scheme, the trust of a ticket can be verified mechanically by the proposed ticket type verification procedure, which checks whether the ticket meets the corresponding ticket type definition managed by the user.

The following section presents a ticket circulation model. The third section describes the circulation control scheme in detail. We then discuss the trust management scheme that is the basis of security in Section 4. Section 5 overviews an implementation of the ticket circulation system. Finally, we draw a comparison to related work in Section 6.

2. Ticket Circulation Model

This section presents the basic ticket circulation model assumed herein. The design goal of *no centralized organization* is the only one related to the basic model and we address it in this section. Approaches to the other design goals are described in Section 3 and 4.

2.1 Participants

The participants in our ticket circulation model and the assumed ticket flow are shown in Figure 1. There are three types of participants in the ticket circulation model: an *issuer* creates, signs, and issues a ticket; a *user* redeems the ticket; and a *service provider* fulfills the service or task represented by the ticket. The issuer and service provider can be the same physical organization. Additionally, a *shop*, *broker*, or other participants exist in real paper ticket circulation but they are not included in the settlement because they are treated as users who buy tickets from issuers (or users) and transfer the tickets to other users with payment.

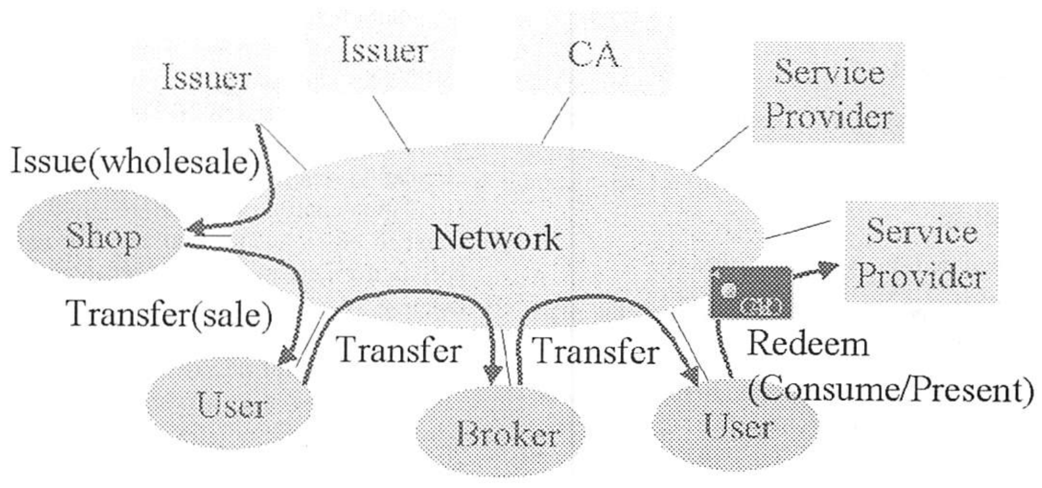


Figure 1. Ticket circulation model

2.2 Digital ticket

In this paper, a *digital ticket* (or *ticket*) is defined as $\text{Signed}_I(I, P, O)$, where I is the ticket issuer, O is the ticket owner, and P is a *promise* to the ticket owner. The phrase "Signed_I" means that the entire block is signed by the issuer's digital signature. Promise P has several sub-properties that represent various rights depending on the application.

2.3 Transaction

Circulation of a digital ticket comprises three types of principal transactions:

Issuance is an action in which issuer I gives ownership of ticket T to user U. In our model, we assume that this transaction is implemented by issuer I creating ticket $T = \text{Signed}_I(I, P, U)$ and sending it to user U.

Transfer is an action in which user U_0 transfers ownership of ticket T to user U_1. In our model, we assume that this transaction is implemented by attaching a *transfer certificate* to the ticket to be transferred, i.e., user U_0 creates transfer certificate $T_{t1} = \text{Signed}_{U0}(U_0, \text{transfer}(T), U_1)$ and sends it to user U_1 with T, where transfer(T) is a promise that T was transferred.

Redemption is an action in which user U redeems the rights represented by ticket T to service provider S. In our model, we assume that this transaction is implemented by attaching a *redeem certificate* to the ticket, i.e., user U creates redeem certificate $T_r = \text{Signed}_U(U, \text{redeem}(T), S)$ and sends it to service provider S with T, where redeem(T) is a promise that T was redeemed. The situation in which ownership of the ticket is retained when the ticket is redeemed, e.g., redemption of licenses or passports, is termed *presentation*. The situation in which ownership of the ticket is voided or the number of times it is valid is reduced when the ticket is redeemed, e.g., redemption of event tickets or telephone cards is termed *consumption*.

Assume that a ticket was circulated between participants $I \rightarrow U_0 \rightarrow U_1 \rightarrow \ldots \rightarrow U_n \rightarrow S$, using the transactions of issuance, transfer, and redemption. A set of tickets T, $T_{t1}, \ldots, T_{tn}$, T_r, called a *transfer list*, is sent to the service provider as a result of the circulation. We use the transfer list to detect who transferred or redeemed a ticket more than once after fraud is detected. Our prototype system also offers an offline fraud prevention scheme using a smart card, but this scheme is beyond the scope of this paper.

Generally speaking, money, services, or products are circulated against the flow of the tickets. There are issues on how the atomicity between these two delivery flows is guaranteed [6]. This paper, however, focuses only on the ticket flow and makes payment methods independent because this approach enables easy integration with legacy application systems that use existing payment systems. Of course, integration with payment methods is an important issue to be studied.

3. Circulation Control Scheme

In this section, we present an approach to satisfy the design goals described in Section 1: *business scheme flexibility*, *management autonomy*, and *simplicity*.

3.1 Requirements

Business scheme flexibility is realized by establishing a general circulation control scheme that can handle various circulation requirements some of which are described below:

(1) Only qualified shops can issue tickets

(2) Tickets may be circulated only between the registered members of a group

(3) Only certain agents are allowed to transfer tickets, the general public is prohibited from transferring tickets to anybody else, e.g., plane tickets and event tickets

(4) Only qualified people can redeem tickets, e.g., student discount tickets

(5) Only a certain shop or agent is allowed to examine (punch) the ticket. Most tickets have such a condition

Management autonomy can not be achieved if the circulation system forces every participant to meet specific requirements, e.g., each participant must be registered with a specific CA. In this case, the management policies would be fixed by the CA's certificate policy. Which certificates are required for a transactions should depend only on the application. The management policy of an application must be application (or ticket) specific and not circulation-system specific.

3.2 Onion ticket accumulation model

We found that the above requirements can be satisfied by checking if the participants of a transactions have certificates confirming their qualifications before conducting the transaction. To represent these qualifications, we introduce *circulation control tickets*. The circulation requirements are specified by what type of circulation control tickets are required for each type of ticket to be circulated, and separate requirements are specified for the sender and receiver. The ticket circulating system issues, transfers, or redeems a ticket only if all circulation requirements are satisfied. The circulation control tickets themselves can be any type of digital ticket, e.g., a driver's license, certain membership certificate, or other certificates issued by a CA, and these tickets can be recursively circulated using the ticket circulating system. We call this scheme the *onion ticket accumulation model* (Figure 2).

In this model, the identity of a participant forms the onion core, and the rights (tickets) given to the identity form the layers beyond the core. A transaction is conducted between two onions (participants). The model illustrates that an outer layer (tickets) cannot be peeled or moved to another onion unless inner layers (tickets) exist.

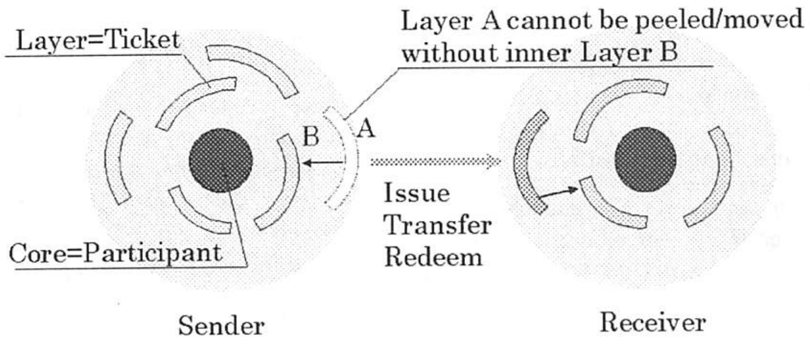


Figure 2. Onion Ticket Accumulation Model

Transactions	Sender conditions	Receiver conditions
Issue	Airline certificate	(None)
Transfer	Travel agent certificate	(None)
Redeem	(None)	Airline certificate

Table 1. Example of Circulation Condition Definition

For example, a plane ticket can be modeled using circulation conditions as shown in Table 1. A plane ticket can be issued and punched by the airlines that have an airline certificate issued by the International Air Transport Association (IATA). A plane ticket can be transferred by the travel agencies who have a travel agent certificate issued by a government, while the public is prohibited from transferring the tickets to anybody else.

The next issue is how and where these circulation requirements are defined in the system. We describe this issue in the following sections.

3.3 Ticket type definition

To make the ticket-circulation system application independent, we propose specifying the circulation requirements for a ticket in the ticket itself using the Generalized Ticket Definition Language. However, efficiently and securely specifying the "type" of control tickets, e.g., airline certificates or travel agent certificates, in the circulation requirements is still an issue.

We found that there are two classes of information in a ticket. One is common to each type of ticket. The other is different for each ticket instance. For example, the circulation requirements presented in Section 3.2 are an instance of the former class of information and do not have to be defined for each ticket. We, therefore, introduce two types of definitions: a *ticket type definition* and *ticket definition*. The common information within tickets of the same ticket type is defined in the ticket type definition. The specific information on the ticket instance is defined in the ticket definition. To bind a ticket definition to its ticket type definition securely, we propose setting the hash value of the ticket type definition in the ticket definition, which is digitally signed by the ticket issuer. This scheme reduces communication cost and improves efficiency of ticket circulation since the ticket type definition can be pre-distributed to the participants who may use the tickets of a particular ticket type. Details of the distribution of the ticket type definitions are described in the following section.

A *ticket type definition* is the following tuple:

TypeInfo: The ticket type definition information. It includes the contact address of the person defining the ticket type, the name of the ticket type, version number, or other information.

TypeValidity: The validity condition of the ticket type. The valid period, start date and end date, are specified.

IssueConditions: Pre-conditions of the issue transaction. This is a tuple of *SenderConditions* and *ReceiverConditions* described below.

TransferConditions: Pre-conditions of the transfer requirement. This is a tuple of *SenderConditions* and *ReceiverConditions* described below.

RedeemConditions: Pre-conditions of the redeem transaction. This is a tuple of *SenderConditions* and *ReceiverConditions* described below.

TicketSchema: Syntax definition for the Promise property of the tickets of this ticket type. This defines sub-properties of the *Promise* property and the necessity value, i.e., mandatory or optional, and default value if any. There are several specifications for this purpose. XML DCD [16] or SOX [19] can be used [12]. Details of the definition method are thus outside the scope of this paper.

SenderConditions and *ReceiverConditions* are one of the following:

(1) $TypeID_1 \ldots TypeID_n$: A list of the ticket type identifiers that must be held by the sender or receiver, where $TypeID_X$ = hash(*the ticket type definition X*).

(2) *Identity$_X$*: The identity of the sender or receiver X. It can be implemented using a public key or other names with the scope rule but this paper assumes *Identity$_X$* = hash (PK_X) for simplicity.

(3) *Not specified*: This means no restrictions apply to the sender or receiver.

Condition (1) specifies what types of tickets the participants must possess and condition (2) specifies the participants directly. Conditions are application dependent and specified by the business scheme or legislation.

In the plane ticket example shown in Section 3.2, an airline certificate is specified as the sender condition within the issue conditions. This is an example of condition (1) above. This condition can be defined by specifying the ticket type identifier of the airline certificate in the type definition of the plane ticket. This scheme enables new airlines to issue plane tickets after getting their airline certificates from IATA. It is not necessary to redistribute ticket type definition to users in this case. As an example of condition (2) above, we assume a plane ticket, the type of which is defined by each airline and issued without IATA involvement. In this case, the sender conditions within the issue conditions can be defined by specifying the airline's identity in the type definition of the plane ticket. This scheme does not require new airlines to get their airline certificate from IATA but they must distribute the ticket type definition to the users.

There are several ways in which sender and receiver conditions can be specified other than (1) and (2) above; disjunction, conjunction, and threshold [3][8] of the ticket types or ticket instances, restrictions on property values of a ticket, etc. can be used. Our analysis of paper tickets shows that implementing conditions (1) and (2) achieves sufficient flexibility for describing the conditions of most tickets.

3.4 Ticket definition

Basically a ticket has the structure Signed$_I$ (I, P, O) as described in Section 2. To establish a binding between a ticket and its ticket type, we introduce a *ticket type identifier* in the ticket definition. Additionally, two other properties, *ticket instance identifier* and *ticket validity*, are also introduced for implementation practicality. The ticket instance identifier is useful for efficiently detecting duplicate redemption.

A *ticket definition* is the following tuple:

TypeID: The ticket type identifier.

TicketID: The ticket instance identifier. It must be unique for each ticket with the same TypeID.

TicketValidity: The validity conditions of the ticket. At least the valid period, start date and end date, are specified.

IssuerID: The ticket issuer's identity.

Promise: Various properties are specified depending on the application.

OwnerID: The ticket owner's identity.

Signature: Issuer's digital signature on the above.

4. Trust Management Scheme

In this section, we present an approach to satisfy the design goal of *trust manageability* described in Section 1.

4.1 Requirements

The proposed ticket circulation system enables various tickets to be issued very easily by using Generalized Ticket Definition Language based definitions without developing software for ticketing and ticket examination servers. As a result, a wide variety of digital tickets may be circulated freely. This makes it difficult for users to judge whether a ticket can be trusted or not. For example, even if the digital signature of the ticket is valid, this is no guarantee of the rights described in the ticket since it might be signed by a person who has no right to issue the ticket. There are two approaches to preventing or detecting forgery:

Trusted broker: A ticket bought directly from a trusted broker can be trusted even if the ticket issuer cannot be trusted. In this scheme, however, several problems exist: no offline capability, centralized organization, and additional payments to the broker.

Authorization by a CA: If a CA who authorizes all issuers for all types of tickets exists, users can ask the CA if the ticket can be trusted. This scheme, unfortunately, destroys management autonomy. If a different CA exists for each specific type of ticket, management autonomy might be achieved but it is difficult for users to find a CA who can judge the ticket. No practical way of managing the binding between a specific type of ticket and its CA has be proposed yet.

The trust management scheme for digital tickets should, therefore, provides some means of verifying if a ticket can be trusted regardless of its circulation route. It also should provide a tool for managing the basis of trust, which might differ with the ticket type.

4.2 Ticket type based trust management

To achieve the above requirements, this paper proposes a new trust management scheme based on the ticket type. In this scheme, once a user establishes the binding between the conceptual rights in the real world and the ticket type definition (or its identifier), the user can mechanically verify the trust of the same type tickets.

We assume that a user establishes a binding between conceptual rights recognized in the real world and a ticket type identifier by some means. We use this binding as the basis of trust desired by the user. Examples of binding are as follows:

Conceptual rights	Ticket type identifier
Plane ticket	F796452E753FFDEE4379BB2C883C2CAA
Lottery ticket	AAE1DC379BB2C883C2CAAF796452E753
Ticket for car wash	85579BB2C883C2CAAF796452E7536651

This scheme similar to PGP [15] in which a set of bindings between user IDs and public keys (or fingerprints) is the basis of trust.

There are several ways to distribute the ticket type identifiers. For example:

- CD-ROMs sold in bookshops.
- Physical ticketing machines managed by service providers.
- A trusted web site with a secure communication channel.

To manage ticket types, which are key elements of trust, this paper introduces a *ticket type book*. A ticket type book is managed by each user and it stores information about the ticket types trusted by the user. In the ticket type book, a type is managed as the tuple (*TypeName*, *TypeID*, *TicketTypeDefinition*). When a new type is given, the tuple is stored in the ticket type book only if it is *trusted* by the user. A *TypeName* represents conceptual rights recognized by the user and any name can be replaced as he/she likes.

Assuming that a user has the ticket type book described above, the user can use the following type verification procedure to verify if the issuer has the right to issue the ticket:

Ticket type verification procedure: Let T be a ticket definition and TT be a ticket type definition. We say that T *meets* TT if and only if the following procedure is completed.

(1) Verify the digital signature of T.

(2) Verify that the ticket type identifier *TypeID* in T and hash(TT) are the same.

(3) Verify that the ticket type identifier *TypeID* is in the ticket type book.

(4) Retrieve TT from the ticket type book.

(5) If identity $Identity_X$ (onion core) is specified as the issuer conditions in TT, verify that $Identity_X$ and the issuer identity *IssuerID in* T are the same.

(6) If ticket type identifiers $TypeID_1 \ldots TypeID_n$ (onion layers) are specified as the issuer conditions in TT, verify that the issuer owns all tickets $T_1 \ldots T_n$ corresponding to $TypeID_1 \ldots TypeID_n$ (getting $T_1 \ldots T_n$ from the issuer directly or from the circulation history attached to the transferred ticket, and verify that *OwnerIDs* in $T_1 \ldots T_n$ and *IssuerID* in T are the same), and verify that for each ticket definition $T_1 \ldots T_n$ *meet* the corresponding type definitions $TT_1 \ldots TT_n$ by applying this procedure recursively.

Note that verification of *TicketValidity* in T and *TypeValidity* in TT, and verification that the *Promise* defined in T meets the *TicketSchema* in TT are not included in the above procedure for readability. The verification of the series of transfer certificates, which must be verified for a transferred ticket, is also omitted for readability.

This trust management scheme enables users to detect a forged ticket regardless of the circulation route, and can be performed mechanically by the proposed type verification procedure. This scheme also provides easy management of the basis of trust since the user need manage only one type definition for each type of ticket, even if complicated issuer conditions are set.

5. Overview of Implementation

5.1 Generalized ticket definition language

We proposed to implement the Generalized Ticket Definition Language on top of RDF [18], which, in our previous paper [9], is layered on XML [17]. Our current implementation, however, uses XML directly [12]. The reason for this is because RDF is too rich and redundant to describe circulation conditions or other properties for controlling ticket circulation, which are common to all tickets, and the semantics are not important except with regard to the Promise property.

The standardization of XML signed documents [5] is now being actively discussed in W3C and IETF. Our current specification does not comply with these specifications but integration with any future standard is an important issue to be studied.

5.2 Protocols

The ticket circulation protocols are overviewed in Figure 3. First, a definition for the ticket to be circulated is sent to the receiver in order to send the circulation conditions. In this phase, ownership is not transferred. Second, the receiver checks if the ticket type of the ticket definition is in the receiver's ticket type book. If not, the receiver obtains the type definition in some way, e.g., from the sender or network. Third, the receiver and sender check if the circulation condition is satisfied. In this phase, tickets that must be owned by the sender or receiver are checked against each other. Finally, ownership is transferred from sender to receiver by sending a transfer or redemption certificate.

We have implemented a prototype of the above protocols using Java and confirmed its feasibility. We also implemented three common components on top of the protocol handling system: ticketing server, ticket examination server, and ticket wallet.

5.3 Application example

The GUI of the ticket wallet is shown in Figure 4. In this example, four types of tickets are issued and stored in the ticket wallet after conducting several transactions in the following scenario: First, a wallet certificate was issued when a user installed his/her ticket wallet. Next, a customer certificate for a loyalty program was issued when he/she registered at the shop. After several points were earned at the shop, an award ticket, which can be exchanged for a prize, was issued with the redemption of some of the earned points.

Based on the onion ticket accumulation model, several requirements can be given in this example such as; the customer certificate requires users to have a wallet certificate and award ticket transfer requires users to have the customer certificate. Such flexible circulation control can be achieved without any application-specific software in the ticket wallet.

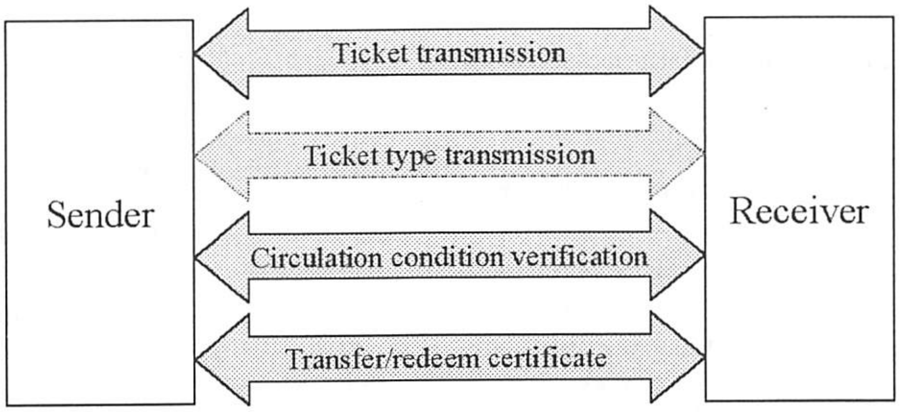
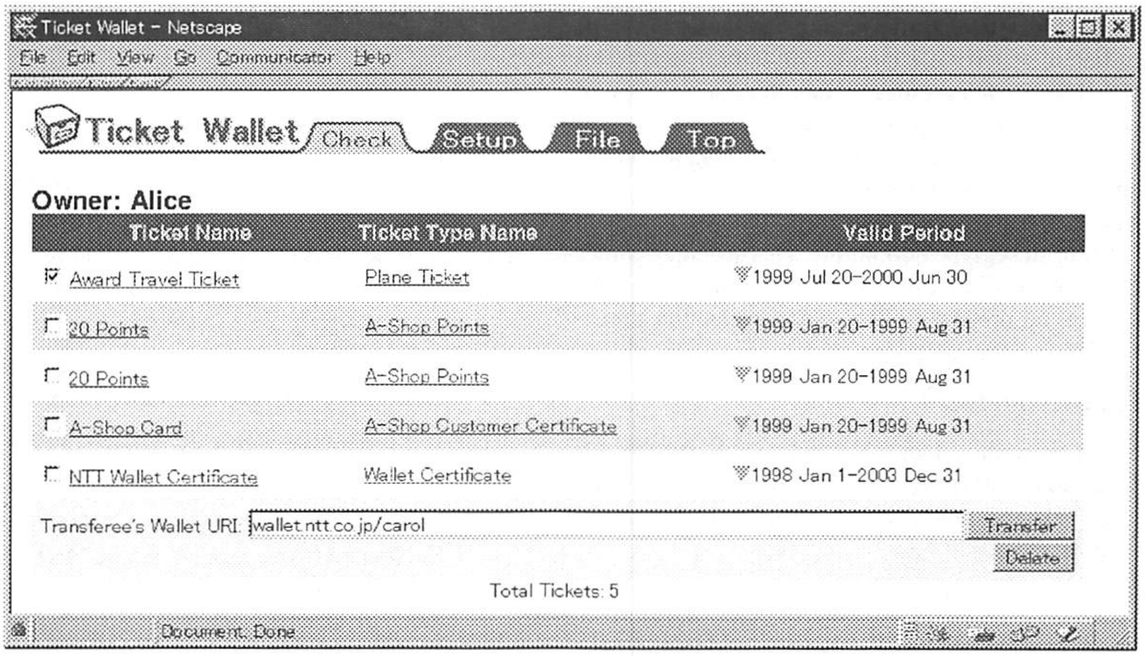

Sender
Ticket transmission
Ticket type transmission
Circulation condition verification
Transfer/redeem certificate
Receiver

Figure 3. Ticket Circulation Protocol Overview



Owner: Alice

Ticket Name	Ticket Type Name	Valid Period
✓ Award Travel Ticket	Plane Ticket	1999 Jul 20-2000 Jun 30
20 Points	A-Shop Points	1999 Jan 20-1999 Aug 31
20 Points	A-Shop Points	1999 Jan 20-1999 Aug 31
A-Shop Card	A-Shop Customer Certificate	1999 Jan 20-1999 Aug 31
NTT Wallet Certificate	Wallet Certificate	1998 Jan 1-2003 Dec 31

Transferee's Wallet URI: wallet.ntt.co.jp/carol

Total Tickets: 5

Figure 4. Ticket wallet displayed in a Web browser

6. Related Work

KeyNote [3][4] presents an authorization control system that determines whether actions are consistent with policies, which are a collection of certificates called assertions. We present here a ticket circulation system that determines whether issuance, transfer, and redemption requirements are consistent with the sender or receiver qualification conditions. In this sense, our approach is similar to but more specific than KeyNote and is focused on ticket circulation. The KeyNote system uses a general programmable language to define assertions, whereas our system uses a ticket type based specific language to allow high-level control descriptions.

SPKI [8] and PGPticket [11] present an authorization control system that provides a mechanism for deriving authorization decisions from a collection of certificates. They provide the ability to delegate fine-grained authorization from one person to another. These systems, however, do not introduce certificate transferability, in which the transferor loses the rights when the certificate is transferred. As a result, it is difficult to realize event tickets or other tickets that can be consumed only once, although we note that it can be applied to license or pass-type tickets.

The Eternal Resource Locator [1] presents a scheme to establish trust without relying on any PKI. This scheme uses the hash value of the root hypertext document as the basis of trust. Our scheme has similarities to this in that the hash value of the type definition is used as the basis of trust.

NetBill [6] presents an authorization scheme, in which a customer's authority is represented by an identity ticket, which is a pseudonym of the customer, and one or more *credentials*, each of which represents proof of group membership. This model has some similarities to our onion ticket accumulation model. The NetBill system introduced this model mainly for flexible price control, whereas our model is used to increase the flexibility of circulation control.

Capability cards [13] represent a digital media that can circulate various types of digital objects including digital rights using a card metaphor. However, they do not provide flexible circulation control or trust management, both of which are offered by the schemes proposed in this paper.

7. Conclusion

This paper described issues and design goals for a generalized ticket circulation system that can circulate the various types of digital tickets required in diverse business schemes. To enable flexible control of ticket circulation, we specify circulation control tickets. Before issuing, transferring or redeeming a ticket, the sender or receiver must have the appropriate circulation control tickets. We define a ticket in two parts, ticket type and individual ticket information, and use the type identifier to specify the circulation control tickets required. These schemes make it possible for any type of ticket, e.g., driver's license or social security certificate, to be used as the PKI for ticket circulation in addition to identity certificates issued by a CA. We also proposed a new trust management scheme based on the trust of ticket type definitions. This scheme enables users to mechanically verify the trust of a ticket by executing the proposed ticket type verification procedure.

Acknowledgement

The authors wish to thank Yoshiaki Nakajima, Yoshihito Oshima, Masayuki Hanadate, Nobuyuki Chiwata, Tomoji Takehisa, and Takaaki Matsumoto for useful comments and discussions.

References

[1] R. J. Anderson and V. Matyáš Jr., "The Eternal Resource Locator: An Alternative Means of Establishing Trust on the World Wide Web", *3rd USENIX Workshop on Electronic Commerce*, August 1998, pp. 141-153.

[2] N. Asokan, P. A. Janson, Michael Steiner, and M. Waidner, "The State of the Art in Electronic Payment Systems", *IEEE Computer*, September 1997, pp. 28-35.

[3] M. Blaze, J. Feigenbaum, and M. Strauss, "Compliance Checking in the PolicyMaker Trust-Management System", In *Proceedings of Financial Cryptography '98*, LNCS 1465, pp. 254-274.

[4] M. Blaze, J. Feigenbaum, A. D. Keromytis, and J. Ioannidis, "The KeyNote Trust-Management System", IETF Internet Draft, August 1998.

[5] R. D. Brown, "Digital Signatures for XML" IETF Internet Draft, January 1999.

[6] B. Cox, D. Tyger, and M. Sirbu, "NetBill Security and Transaction Protocol", *1st USENIX Workshop on Electronic Commerce*, July 1995.

[7] C. M. Ellison, "Establishing Identity Without Certification Authorities", *6th USENIX Security Symposium*, July 1996.

[8] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen, "SPKI Certificate Theory", IETF Internet Draft, November 1998.

[9] K. Fujimura and Y. Nakajima, "General-purpose Digital Ticket Framework", *3rd USENIX Workshop on Electronic Commerce*, August 1998, pp. 177-186.

[10] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro, "The Millicent Protocol for Inexpensive Electronic Commerce," In *Proceedings of WWW4*, December 1995.

[11] V. Moscaritolo and A. N. Mione, "PGPticket", IETF Internet Draft, November 1998.

[12] Y. Nakajima and K. Fujimura, "XML Ticket Model and Syntax Specification," IETF Internet Draft, to appear.

[13] K. Otani, H. Sugano, and M. Mitsuoka, "Capability Card: An Attribute Certificate in XML", IETF Internet Draft, November 1998.

[14] Peter Wayner, "Digital Cash", Academic Press Ltd., 1997.

[15] P. Zimmermann, The Official PGP User's Guide, MIT Press, 1995.

[16] Document Content Description for XML (DCD), The World Wide Web Consortium, Note, 1998, http://www.w3.org/TR/NOTE-dcd

[17] Extensible Markup Language (XML) 1.0, The World Wide Web Consortium, Recommendation, 1998, http://www.w3.org/TR/REC-xml

[18] Resource Description Framework (RDF) Model and Syntax Specification, The World Wide Web Consortium, Recommendation, 1998, http://www.w3.org/TR/REC-rdf-syntax/

[19] Schema for Object-oriented XML (SOX), Note, 1998, http://www.w3.org/TR/NOTE-SOX/

[20] ISO/IEC 9594-8 (X.509), Information Technology - Open System Interconnection - The Directory: Authentication Framework

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight–ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from Academic Press, Morgan Kaufmann, New Riders/Cisco Press/MTP, O'Reilly & Associates, OnWord Press, The Open Group, Sage Science Press, and Wiley Computer Publishing.
- Special subscription rates for Cutter Consortium newsletters, *The Linux Journal, The Perl Journal, IEEE Concurrency, Server/Workstation Expert*, *Sys Admin Magazine*, and all Sage Science Press journals.

Supporting Members of the USENIX Association:

C/C++ Users Journal
Cirrus Technologies
Cisco Systems, Inc.
CyberSource Corporation
Deer Run Associates
Greenberg News Networks/MedCast Networks
Hewlett-Packard India Software Operations
Internet Security Systems, Inc.
JSB Software Technologies
Microsoft Research
MKS, Inc.
Motorola Australia Software Centre
NeoSoft, Inc.
New Riders Press
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Questra Consulting
Sendmail, Inc.
Server/Workstation Expert
TeamQuest Corporation
UUNET Technologies, Inc.
Windows NT Systems Magazine
WITSEC, Inc.

Sage Supporting Members:

Atlantic Systems Group
Collective Technologies
D. E. Shaw & Co.
Deer Run Associates
Electric Lightwave, Inc.
ESM Services, Inc.
GNAC, Inc.
Mentor Graphics Corp.
Microsoft Research
MindSource Software Engineers
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
RIPE NCC
SysAdmin Magazine
Taos Mountain
TransQuest Technologies, Inc.
Unix Guru Universe

For further information about membership, conferences or publications, contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738.
Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-28-6